Universidad Carlos III de Madrid

Escuela Politécnica Superior

Departamento Tecnología Electrónica

Trabajo Fin de Grado

Grado en Ingeniería en Tecnologías Industriales

IMPLEMENTACIÓN DEL ALGORITMO DE CIFRADO AES MEDIANTE GPUS DE BAJO COSTE

Autora: María Rodríguez Sánchez

Tutor: Luis Mengibar Pozo

Septiembre 2014

Agradecimientos

En primer lugar quiero dar las gracias a mis padres y hermanos, por ayudarme y apoyarme durante estos años de estudio y por luchar contra mi negatividad día tras día. Porque sin su ayuda no hubiese sido posible la finalización de tantos años de estudio y mucho menos la realización de este trabajo.

También me gustaría acordarme de mis compañeros de clase, por todas las horas que hemos compartido en clases y laboratorios, sin los que tampoco la universidad hubiese sido lo mismo.

Además quiero agradecer a mi tutor, Luis Mengibar Pozo, por todo lo que me ha enseñado durante los meses de ejecución de este TFG, y porque siempre estaba dispuesto a ayudarme.

Resumen

El presente Trabajo Fin de Grado consiste en la implementación del algoritmo de encriptado Advanced Encryption Standard (AES), que también se conoce como Rijndael para 128 bits. De este modo, se puede acercar así al mundo de la encriptación a cualquier persona, ya que estas técnicas están cada día más presentes en nuestras vidas.

AES es un algoritmo simétrico y aunque se ha demostrado que no es inmune frente a todos los ataques, si es altamente seguro si no se conoce su correspondiente clave de cifrado. Este algoritmo realiza múltiples operaciones sobre el estado a codificar, aunque son muy repetitivas. Lo que implica que para su ejecución se debe tener una gran capacidad de cálculo, así que se debe intentar reducir el tiempo de ejecución lo máximo posible ya que este hecho puede ralentizar en exceso la ejecución del algoritmo.

En este trabajo, el algoritmo se implementa de dos modos diferentes. Primero utilizando la potencia del ordenador (CPU) y un lenguaje tradicional de programación como es el caso del lenguaje C. Más tarde se utiliza la potencia de la tarjeta gráfica del ordenador (GPU) y el lenguaje C con extensiones CUDA, paralelizando así el proceso. De esta manera se busca reducir el tiempo de ejecución del proceso

A su vez también se analiza brevemente la historia de la encriptación y sus hitos más importantes, haciendo especial hincapié en la aparición del AES. Del mismo modo se estudia el funcionamiento y partes del AES tanto para codificar como para descodificar.

Palabras clave: algoritmo, AES, CUDA, GPU

Abstract

This Bachellor Thesis consists on the implementation of the encryption algorithm Advanced Encryption Standard (AES), which is also know as Rijndael for 128 bits. In this way, it can be brought the world of encryption to anyone, because these techniques are ever more present in our lives.

AES is a symmetric algorithm and although it has been shown that it is not immune to all atacks, if it is highly safe if the encryption key is unknow. This algorithm makes multiple operations on the encode state, although they are very repetitive. This implies, that for the implementation, we must have a big computational power, so we should try to reduce the execution time as much as posible as it can slow excessively execution of the algorithm.

In this work, the algorithm is implemented in two different ways. First using the power of the computer (CPU) and C programming language. Later using the power of computer graphics card (GPU) and CUDA C language to parallelize the process. In this way, we want to reduce the execution time of the process.

At the same, this Bachellor Thesis also discusses the history of encryption and its major events, with special emphasis on the appearance of AES. In the same way, the operation and parts of AES are studied for encryption and decryption.

Keywords: algorithm, AES, CUDA, GPU

Índice de Contenidos

Índice de figuras

Índice de tablas

Lista de acrónimos

AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard New Instructions
AMD	Advanced Micro Devices, Inc
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
BSS	Basic Service Set
CBC	Cipher – Block Chaining
CCN	Centro Criptológico Nacional
CFB	Cipher Feedback
CNI	Centro Nacional de Inteligencia
CPU	Central Processing Unit
CTR	Counter
CUDA	Compute Unified Device Architecture
ECB	Electronic CodeBook
ECC	Elliptic Curve Cryptosytem Elliptic
DEA	Data Encryption Algorithm
DES	Data Encryption Standard
FIPS	Federal Information Processing Standards
GF	Galois Field
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
INTECO	Instituto Nacional de Tecnologías de la Comunicación
IPsec	Internet Protocol security
ISO	Organización Internacional de Normalización
LOPD	Ley Orgánica de Protección de Datos de Carácter Personal
LPI	Ley de Propiedad Intelectual
LSB	Least Significant Bit

LSSI-CE	Ley Servicios de la Sociedad de la Información y Comercio Electrónico
MSB	Most Significant Bit
NIST	Instituto Nacional de Normas y Tecnología
NSA	National Security Agency
OFB	Output Feedback
OpenACC	Open Accelerators
OpenCL	Open Computing Language
PFC	Proyecto Fin de Carrera
RAE	Real Academia Española
RAM	Random-Access Memory
RFC	Request for Comments
RSA	Rivest, Shamir y Adleman
S-BOX	Matriz de Sustitución de Bytes
SIMT	Single Instruction Multiple Threads
SIMD	Single Instruction Multiple Data
TFG	Trabajo Fin de Grado
WLAN	Wireless local area network
WPA2	Wi-Fi Protected Access 2
X-OR	OR-Exclusiva

Capítulo 1. Introducción y objetivos

1.1. Introducción

La sociedad siempre ha estado muy preocupada en ser capaz de guardar y ocultar los secretos de los ojos de todo aquel que no fuese bienvenido en ellos. Esto es así desde el principio de los tiempos, mucho antes del espionaje en la Segunda Guerra Mundial o de los últimos avances en informática, que han convertido la seguridad informática en un factor tan importante para todos nosotros.

Aunque es hoy en día cuando estos temas conciernen al más alto nivel a toda clase de personas y no solo a gobiernos o grandes autoridades, como ocurría hace apenas unos pocos años. Ya que ha sido tras la aparición de las nuevas tecnologías y de las aplicaciones que ello conlleva, cuando la encriptación se ha convertido en un asunto sumamente importante.

Algunas de estas aplicaciones son las múltiples opciones que nos aporta la aparición de internet, por ejemplo, privacidad en los datos almacenados, operaciones bancarias, datos sensibles, compras online, etc. Pero no solo es esencial en este tipo de operaciones, también utilizamos la encriptación a diario y casi sin apreciarlo para mantener nuestra privacidad, por ejemplo al conectarnos a una red WIFI. Si no existiesen esas medidas de seguridad, esos datos podrían ser descubiertos en cualquier momento y por cualquier persona con unos ciertos conocimientos informáticos, pudiendo utilizarlos incluso para cometer actos delictivos vulnerando gravemente nuestra intimidad.

Así pues la seguridad informática es vital para nuestra sociedad y nuestros avances tecnológicos. Por este motivo a lo largo de los años han surgido una serie de algoritmos capaces de mantener nuestra privacidad, o por lo menos lo intentan, ya que cada vez que aparece uno nuevo también lo hacen los múltiples ataques que recibirá a lo largo de su vida útil. Este es el caso del algoritmo Advanced Encryption Standard (AES), que es objeto del presente Trabajo Fin de Grado. AES es un algoritmo de cifrado por bloques, que en 2001 y tras ganar un certamen fue aceptado como estándar por los Estados Unidos. Este algoritmo es altamente seguro pero no es inmune frente a todos los ataques posibles.

En lo referente a seguridad informática, tan importante es tener un buen algoritmo que nos aporte seguridad como que este algoritmo sea rápido, ya que si un algoritmo tarda mucho tiempo en ejecutarse en realidad no puede ser utilizado, porque se ralentizan tanto las operaciones que su aplicación no es viable en la vida real.

Actualmente y debido a las limitaciones técnicas y térmicas que existen en los procesadores no se puede reducir el tiempo de cómputo tanto como sería deseable, por lo que el tiempo de ejecución no puede disminuirse más allá de una serie de optimizaciones en el algoritmo. Ya que nos enfrentamos a día de hoy a una barrera de la técnica.

Una buena solución para reducir el tiempo de ejecución de los algoritmos es utilizar la GPU del ordenador, es decir la unidad de procesamiento gráfico del ordenador que permite paralelizar la ejecución de los algoritmos de cálculo, lo que busca reducir notablemente los tiempos de ejecución. En concreto, gran parte de las tarjetas gráficas que ha fabricado Nvidia en los últimos años permiten utilizar su potencia para paralelizar los algoritmos. Este hecho permite que un gran número de usuarios puedan disponer de esta tecnología a bajo coste, ya que para poder usarla no hace falta un nuevo equipo que suponga un gran desembolso, porque en muchos ordenadores puede venir incluso incluida de serie y no por ello tienen un mayor coste asociado. Este hecho ha permitido acercar este tipo de técnicas a cualquier programador con un mínimo de conocimientos en programación.

Para programar en CUDA se puede programar a partir de lenguajes de alto nivel como son Fortran, C o C++. Se programa en estos lenguajes y luego se hacen pequeñas modificaciones donde se quiere paralelizar el código, es decir, en aquellos lugares donde la carga de trabajo es mayor y por lo tanto la ejecución es más lenta. Además también se puede utilizar OpenACC como estándar abierto que hará aún más fácil la introducción de esas modificaciones. Así pues, esto también acercará esta metodología a un mayor número de personas que si el lenguaje de programación fuese uno nuevo más complicado, que no partiese de uno ya existente ampliamente conocido por cualquier programador.

1.2. Motivación personal

Se decide realizar este proyecto porque supone el complemento perfecto para finalizar los estudios del Grado en Ingeniería en Tecnologías Industriales. Debido a que gracias a este TFG se pueden ampliar los conocimientos en un gran número de materias, y ese es el principal objetivo de un TFG, ampliar y reforzar contenidos relacionados con tu grado.

Además se considera que los temas que trata son actuales, pero también bastante novedosos. Porque actualmente AES es el algoritmo estándar de cifrado, además la tecnología de Nvidia para controlar la potencia GPU es muy reciente, por lo que aún hoy se considera que se encuentra en expansión.

Por estos motivos, se elige este proyecto y no otro. Además también, en mi caso, permite reforzar una serie de debilidades personales que pueden ser esenciales en un futuro laboral no muy lejano. Estas limitaciones se explican como objetivos en el siguiente apartado.

1.3. Objetivos

A la hora de realizar este TFG, se debe pensar en una serie de objetivos personales que se van a ir cumpliendo según se avance en su realización. Este punto es muy importante, si tenemos en cuenta los escasos conocimientos que se tenían de todo lo relacionado con la programación. Porque que mi único contacto con este tipo de programación fue en primer año de carrera y el único lenguaje que conocía era Fortran.

A continuación se explican brevemente los objetivos que se han desarrollado durante la ejecución de este TFG:

- Comprender el funcionamiento del AES en su versión estándar reconocida por el NIST.
- Conocer los conceptos básicos del lenguaje C, así como conseguir programar el código con un cierto grado de dificultad y avance.
- Aprender el uso de Visual Studio como herramienta de desarrollo.
- Formarse en el concepto de programación paralela, con todos los conceptos que ello implica.
- Aprender a usar CUDA y la GPU para paralelizar algoritmos, haciendo especial hincapié en los traspasos de memoria entre la CPU y la GPU.

Como se puede comprobar, los objetivos de este trabajo fin de grado destacan por la evolución desarrollada y los conocimientos adquiridos durante su realización.

1.4. Estructura del documento

A continuación, se describe brevemente el contenido de los diez capítulos que conforman el presente Trabajo Fin de Grado:

- El capítulo 1 corresponde a la presentación, en él se realiza una introducción a la importancia de los algoritmos de cifrado. Además se indican la motivación personal y los objetivos.
- En el capítulo 2 se desarrolla el llamado estado del arte. Se explica el concepto de criptografía así como su historia y evolución brevemente. Además se analiza la irrupción del algoritmo AES en el mundo de la criptografía, también se analiza su funcionamiento y su base matemática para el caso estándar.
- A lo largo del capítulo 3 se comenta el entorno legislativo, es decir, el marco regulador sobre el cual se puede aplicar AES, algo esencial en la seguridad informática.
- El capítulo 4 se utiliza para detallar las características del entorno de trabajo, es decir, principalmente se comentan las especificaciones del PC y la GPU.
- El capítulo 5 está dedicado a las implementaciones en C, así pues se comentan algunas características de este lenguaje y se detalla el código programado.
- A lo largo del capítulo 6 se explica el funcionamiento de la GPU y de la programación en CUDA que permite paralelizar los algoritmos. Se hace especial hincapié en el kernel.
- En el capítulo 7 se comparan ambas implementaciones y sus tiempos de ejecución. Se muestran los todos los resultados obtenidos en los diferentes experimentos, así como el tratamiento aplicado a los datos.
- El capítulo 8 está dedicado a las conclusiones obtenidas mediante los datos del capítulo anterior.
- En el capítulo 9 se detallan los trabajos futuros que se pueden desarrollar en un futuro teniendo de base este TFG, utilizándolo así para ampliar conceptos relacionados con él.
- Finalmente y para concluir en el capítulo 10, se elabora un presupuesto donde se detallan todos los conceptos que se han desarrollado en el mismo y el tiempo necesario para la elaboración de cada uno. Contabilizándose así el tiempo empleado en él y el valor monetario de ese tiempo.

Capítulo 2. Estado del arte

2.1. Criptografía

En esta sección se definen los conceptos esenciales para comprender la criptografía. Además también se comentan brevemente los acontecimientos más importantes en la historia de la criptografía, así como su importancia en el mundo actual.

2.1.1. ¿Qué es la criptografía?

La mejor manera para comenzar a responder esta cuestión es con su significado etimológico, **criptografía** proviene de la unión de las siguientes palabras griegas "*κρυπτός"* que significa oculto y del sufijo *"-grafía"* o *"-γραφία"* que tiene los siguientes significados tratado, escritura, descripción y representación gráfica [1]. De este modo el significado etimológico ya muestra la idea que la **criptografía** es la escritura oculta.

Otra manera de explicar el significado de **criptografía** es con la definición del diccionario de la lengua española de la Real Academia Española. Según la RAE [1], criptografía significa *"Arte de escribir con clave secreta o de un modo enigmático"*.

La **criptografía** no es algo moderno ya que desde hace muchos siglos se ha intentado alterar la información y los mensajes a través de códigos y claves para ocultarlos y hacerlos ininteligibles a otras personas. Aunque es hoy cuando la **criptografía** juega un papel esencial en nuestras vidas, ya que tiene un papel crucial para mantener nuestra seguridad y privacidad. Es esencial en la informática y en todas las aplicaciones que ella nos aporta, pero también gracias a la informática y a las matemáticas la criptografía ha sufrido un nuevo auge.

Como la **criptografía** siempre va asociada a una clave secreta, el mensaje solo podrá ser descifrado por aquellos a los que vaya destinado, de este modo si el mensaje es interceptado por otra persona, no podrá entenderlo si no dispone de la clave necesaria, preservando así el mensaje.

Así pues constantemente y desde la antigüedad han ido apareciendo multitud de algoritmos y sistemas de cifrado, que tenían en su mayoría una fuerte base matemática, estos códigos han permitido ocultar la información de un modo seguro. Esta evolución a lo largo de la historia ha permitido a su vez llegar al nivel de seguridad que actualmente se tiene.

2.1.2. ¿Qué es el criptoanálisis?

Al igual que en el punto anterior, el mejor modo para comprender qué es en realidad el **criptoanálisis** es estudiando su etimología. **Criptoanálisis** proviene de las siguientes palabras griegas "*κρυπτός"* que como ya se ha comentado significa oculto y de *"ἀνάλυσις"* que significa análisis [1]. Así pues el significado etimológico ya muestra la idea de que el principal objetivo del criptoanálisis es analizar lo oculto para lograr hacerlo inteligible.

Además, si atendemos a la definición que aparece en el diccionario de la RAE [1], **criptoanálisis** se define como *"Arte de descifrar criptogramas"*. Así pues, del mismo modo que la criptografía se encarga de ocultar el mensaje, el **criptoanálisis** realiza la función inversa, ya que estudia como descifrar los documentos cifrados.

El **criptoanálisis** estudia los criptogramas, ya que busca las debilidades en los algoritmos de cifrado, para poder llegar a comprenderlos. Así el criptoanálisis intenta burlar la seguridad que ofrecen los algoritmos de cifrado.

De este modo, ambas disciplinas siempre han estado muy unidas a lo largo de la historia. La evolución de ambas es inherente, porque desde el momento en el cuál aparece un nuevo

método de cifrado, también se empieza a analizar intensamente, para poder lograr comprender su funcionamiento y poder así descifrarlo.

2.1.3. Principales acontecimientos de la historia de la criptografía

La **criptografía** ha sufrido una fuerte evolución a lo largo de los siglos. Aunque no lo parezca esta evolución es muy importante en la vida actual y en la concepción que actualmente tenemos de la criptografía, ya que en muchas ocasiones esos sistemas de cifrado de la antigüedad pueden ser considerados la base de los actuales.

Para entender esta evolución es necesario dividir la historia de la criptografía en diferentes partes. Estos bloques son lo que se conoce como la criptografía clásica y la criptografía moderna, además también se hace referencia al estado de la criptografía en la actualidad. En este apartado se señalan brevemente los principales acontecimientos y características que componen cada una de las partes de su historia.

2.1.3.1. Criptografía clásica

Como ya se ha comentado la criptografía ha existido desde el origen de las civilizaciones, ya que siempre ha sido importante para el ser humano ser capaz de ocultar sus secretos. A continuación, se explica brevemente el funcionamiento de los primeros sistemas de cifrado.

Quizás el primer ejemplo claro de un sistema de cifrado sea la **escitala** espartana. La **escitala** apareció por primera vez en el siglo IV a. C. [2], consistía en un cilindro de madera de un determinado diámetro y una tira de cuero. El mensaje se escribía en esta tira, pero solo se podía leer correctamente si se tenía otro cilindro gemelo con el mismo diámetro, así pues esta era la clave que debía conocer el receptor. La **escitala** utilizaba la trasposición como método de cifrado. Esto permitía que si la cinta era interceptada y no se conocía el diámetro exacto del cilindro, el mensaje solo parecía un conjunto de letras sin sentido.

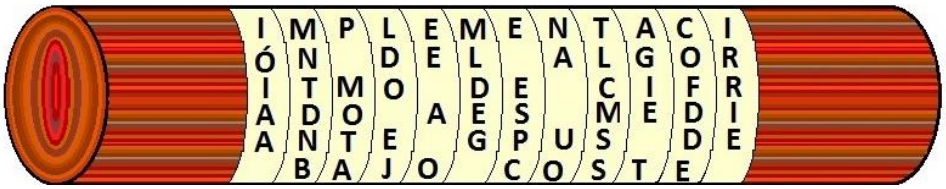

Figura 2.1 La escitala espartana

Hacia el siglo II a. C. se comenzó a usar el **cifrador** o **tabla de Polybios** [3]. Este método utilizaba la sustitución, ya que consistía en una tabla como la anexa. Como se puede comprobar esta tabla asigna a cada letra del mensaje a cifrar un nuevo par de letras, formando así el criptograma.

Tabla 2.1 Cifrador de Polybios [2]

	A	B	C	D	E
A	A	B	C	D	E
B	F	G	H	I,J	K
C	L	M	N	O	P
D	Q	R	S	T	U
E	V	W	X	Y	Z

Su funcionamiento era sencillo, por ejemplo si se quiere cifrar la letra G, con la tabla se convierte en BB. Otra versión de esta técnica consiste en sustituir las letras A, B, C, D, E por los números 1, 2, 3, 4, 5 para cifrar [4]. Por lo demás su funcionamiento es totalmente análogo.

Este método puede ser considerado bastante simple porque sólo utiliza cinco caracteres para encriptar.

Uno de los algoritmos que hoy en día aún se recuerda más ampliamente es el **Cifrado César**. Esta técnica se utilizó en el siglo I a. C. [3], consistía en sustituir cada letra por aquella que se sitúa tres posiciones más a la derecha en el abecedario. De este modo, por ejemplo la letra G se transforma en la J con el **Cifrado César**. Este método era simple y tenía varias debilidades, porque se podían apreciar fácilmente la repetición de caracteres en los mensajes cifrados [4], lo que podía dar señas de su funcionamiento.

Ya en el siglo XV, concretamente en 1466, Leon Battista Alberti [5] creó el que se considera el primer cifrador polialfabético. El **disco de Alberti** está formado por dos discos concéntricos. El externo es fijo y se utiliza para cifrar, contiene los números del 1 al 4 y los 20 caracteres ordenados del alfabeto en latín, sin incluir la H, J, K, Ñ, U, W e Y. El disco interno es móvil, contiene 24 casillas y se usa para el texto cifrado, está formado por los 20 caracteres del alfabeto latín desordenados, el signo & y las letras H, K e Y.

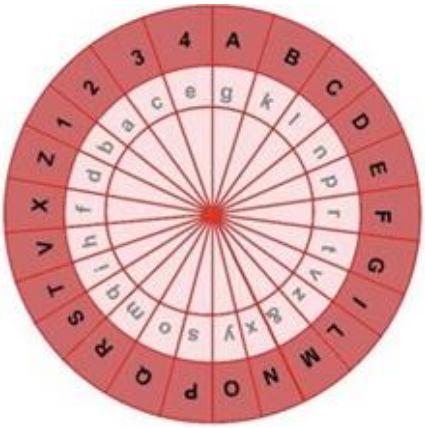

Figura 2.2 El disco de Alberti [2]

Al ser móvil el disco interno se debe fijar en él una posición para el cifrado, esta será la clave que el receptor debe conocer. Además también es posible cambiar la posición de cifrado cada cierto número de caracteres, incluso se puede considerar que existen 24 alfabetos de cifrado, tantos como posiciones tiene el disco. Esto supuso una auténtica revolución para su época.

Otros métodos de cifrado que están relacionados o que provienen en parte del disco de Alberti son los siguientes:

- El **cifrado de Vigenère** (siglo XVI) que utiliza la base del disco de Alberti pero en forma de tabla compuesta por los 26 alfabetos del César para cifrar.[2][4]
- El **cifrado de Beaufort** (siglo XVIII), muy similar al de Vigenère porque utilizaba la misma tabla pero del modo inverso.[2][4]
- La **rueda de Jefferson** (siglo XIX), formado por una serie de discos que pueden girar libremente sobre un eje, cada disco lleva el alfabeto escrito en un orden distinto, así el emisor desplaza los discos para componer el mensaje que se quiere transmitir y el cifrado. [5]
- El **disco de Wheatstone** (siglo XIX), utiliza un ingenio muy parecido al disco de Alberti pero con el espacio y las 26 letras del alfabeto inglés en el disco exterior y solo las 26 letras en el disco interior. Además tiene dos agujas, similares a las de cualquier reloj, engranadas de tal forma que por cada vuelta de la aguja grande, la pequeña recorre esa vuelta más una letra, produciéndose así un cierto desfase.[4] [6]

Aunque hoy en día algunos de estos métodos de cifrado de la antigüedad nos puedan parecer muy rudimentarios, en realidad eran muy ingeniosos para la época en la que se inventaron. Además eran bastantes difíciles de descifrar, si se tienen en cuenta los conocimientos que se tenían en aquel momento. Del mismo modo, al igual que la criptografía, el criptoanálisis no estaba tan avanzado como en la actualidad y con cada nuevo ingenio también el criptoanálisis daba un paso hacia delante para llegar a su posición actual.

2.1.3.2. Criptografía moderna

Poco a poco la criptografía fue avanzando y se empezó a convertir en algo muy importante para todas las naciones, especialmente en todo lo relacionado con los asuntos bélicos. Aunque no fue hasta el siglo XX, cuando la criptografía realmente asumió un papel esencial en la historia. Tras la Primera Guerra Mundial, los gobiernos se dieron cuenta de la importancia de la criptografía en el arte de la guerra y especialmente en sus comunicaciones, pero la Segunda Guerra Mundial fue la que realmente supuso una auténtica revolución en la historia de la criptografía.

En el siglo XX volvieron a aparecer sistemas de cifrado, a continuación se explican brevemente algunos de estos sistemas. El **Cifrado Vernam** (1917)**,** que consistía en un sistema de sustitución mediante un alfabeto binario de cinco dígitos por cada letra y una clave binaria aleatoria [2]. La **Cifra ADFGVX** (1918), este sistema usaba una cuadrícula de 7x7 que se completaba con las letras ADFGVX, las letras del abecedario y los 10 números de manera aleatoria, además esta disposición estaba relacionada con la clave, más tarde había que realizarle una nueva transformación [2]. Estos métodos fueron ampliamente utilizados durante la guerra y fueron considerados altamente seguros y difíciles de descifrar.

Aunque lo que realmente supuso un gran cambio en la concepción de la criptografía fue la aparición de las máquinas de cifrar formadas por rotores. Estas máquinas automatizaban y aceleraban mucho los cálculos en el proceso de cifrado. Se desarrollaron en el siglo XX, pero fue durante la Segunda Guerra Mundial cuando se convirtieron en protagonistas.

La máquina de este tipo más importante y también la más recordada es la **Máquina Enigma** (1918), debe su notoriedad a que fue ampliamente utilizada por los alemanes durante la Segunda Guerra Mundial. Consistía principalmente en un teclado (similar al de una máquina de escribir) donde se escribía el mensaje, la unidad modificadora que estaba formada por un banco de rotores con 26 contactos eléctricos en su perímetro [4] y el tablero donde aparecía el mensaje cifrado. Además para descifrar el mensaje existía otro elemento llamado Reflector, que permitía conocer el texto.

Figura 2.3 Máquina Enigma

Para aumentar la seguridad del cifrado, se fueron añadiendo más unidades modificadoras, que incluso se podían variar de posición. Por si esto fuera poco variaban la clave de cifrado cada 24 horas.

Finalmente la máquina fue descifrada gracias a la intervención de un gran número de científicos y personalidades dedicados al criptoanálisis, entre los cuales destaca la labor de Alan Turing. Su análisis según algunos expertos permitió la conclusión de la guerra dos años antes. [7]

Posteriormente la criptografía siguió evolucionando hasta su posición actual, por ejemplo la posición que tienen los ordenadores actuales y los complejos algoritmos de cálculo en la criptografía. Aunque se puede considerar que en parte esta máquinas de rotores de principios del siglo XX son la base de la criptografía actual.

2.1.3.3. Criptografía en la actualidad

Como ya se ha comentado es hoy en día cuando la criptografía es un asunto que incumbe a todas las personas, es decir, hoy la criptografía no es solo un asunto de espionaje, es más una cuestión que importa a toda la sociedad. Esto es así debido a la aparición de los sistemas informáticos en las comunicaciones y al uso de internet de manera habitual.

Este hecho ha provocado que la criptografía sea esencial para mantener la seguridad informática, así pues la criptografía está presente en la gestión de certificados, firma electrónica, compras online, correo electrónico, etc.

En el siglo XX también han aparecido nuevas técnicas que utilizan otros métodos para cifrar, este es el caso por ejemplo de la **criptografía de curva elíptica (ECC)** y de la **criptografía cuántica**. La **ECC** utiliza los fundamentos de las curvas elípticas para cifrar, este método es más rápido y permite utilizar claves de menor tamaño para obtener la misma seguridad [3]. Mientras que la **criptografía cuántica** usa los principios de la mecánica cuántica y en el principio de incertidumbre de Heisenberg [3].

Un algoritmo que merece ser destacado es el **RSA**, que es un algoritmo de clave pública creado en 1977, su seguridad viene dada por la dificultad que implica factorizar números grandes. Además utiliza dos claves, una pública y otra privada [8].

En este siglo también han aparecido una serie de algoritmos que se han definido como estándar por alguna institución. De este modo apareció el algoritmo **Data Encryption Standard (DES)**, que más tarde sería conocido como DEA tras pasar una serie de normas oficiales, consistía en un cifrado por bloques al que se le aplicaban una serie de permutaciones y sustituciones [9]. Más tarde le sucedería el **Advanced Encryption Standard (AES)** y que es el objeto de este trabajo, por lo que más tarde se desarrollará ampliamente.

Aunque al principio la gestión de la criptografía, solamente era llevada a cabo por parte de las agencias nacionales y todo su desarrollo se realizaba en secreto. Cada vez es más común que organismos externos también se dediquen al estudio de la criptografía, este es por ejemplo el caso de muchas universidades que realizan sus avances y más tarde los hacen públicos, esto permite que todas las personas se puedan beneficiar de ellos para proteger su información [10].

Además esto conlleva que ambos sectores se encuentren en una cierta competición constante. También existe una gran duda respecto a publicar el funcionamiento de los algoritmos de

cálculo, ya que una parte de la comunidad es contraria a ello, mientras otra piensa que es mejor publicarlos y que toda la comunidad pueda buscar sus debilidades.

2.1.4. ¿Qué es un criptosistema?

El mejor modo para comprender el significado de **criptosistema** es con la definición que da el Centro Criptológico Nacional (CCN). Según el CCN, un **criptosistema** es un *"Conjunto de claves y equipos de cifra que utilizados coordinadamente ofrecen un medio para cifrar y descifrar"* [11]. De esta manera un **criptosistema** es el conjunto de algoritmos que se utiliza para codificar y descodificar un mensaje. A continuación, se muestra un esquema que explica fácilmente el proceso de cifrado y descifrado.

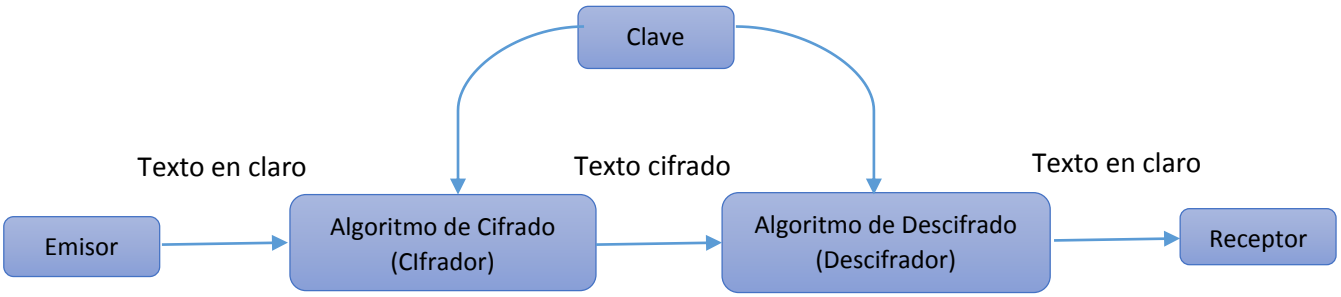

Figura 2.4 Esquema de un criptosistema [12]

Así pues y como muestra el esquema, el emisor cifra, a través de una clave y con la ayuda de un algoritmo, el mensaje que quiere transmitir a la vez que protege su contenido. Una vez que el mensaje llega a su destino, el receptor lo traduce con el algoritmo inverso y la clave, para poder hacerlo inteligible.

De este modo, el objeto de este trabajo además de implementar un algoritmo mediante GPU, es analizar el funcionamiento del algoritmo AES para cifrar y descifrar. Por tanto y tras analizar la definición de criptosistema, en este proyecto se estudia el funcionamiento del criptosistema AES.

También existen diferentes tipos de cifrado, estos se pueden clasificar de diferentes maneras según distintos criterios, a continuación se explican brevemente cada una de ellas y los tipos que hay en cada uno [13]:

Según su **naturaleza**, es decir, según la manera en la que el criptosistema opera para cifrar. Así los sistemas pueden ser de sustitución si siempre sustituye por los mismos datos, de trasposición cuando se permutan los datos o de producto si influye la posición que ocupan.

Otra clasificación es en función del **número de símbolos cifrados con cada clave**, distingue entre que se cifre un único símbolo o varios. De este modo los criptosistemas pueden ser de bloque cuando se cifran varios símbolos para cada clave o de flujo cuando se cifra un único símbolo por clave.

La última clasificación es según la **clave utilizada**, ya que se puede utilizar una clave distinta para cifrar y descifrar, así tenemos criptosistemas simétricos y asimétricos. Así pues los sistemas simétricos son los que utilizan la misma clave para cifrar y descifrar y los asimétricos utilizan dos claves una pública y otra privada.

En este caso el algoritmo AES, se podría clasificar como un criptosistema simétrico y de bloque, si atendemos respectivamente a la clave utilizada y al número de símbolos cifrados con cada clave.

2.1.5. Relación Criptosistema – Criptoanálisis

En todos los avances que se producen en la criptografía siempre ha existido un fuerte compromiso entre los **criptosistemas** y el **criptoanálisis.** Esto es debido a que la relación que existe entre ellos es inherente.

Por ejemplo, se podría crear un nuevo algoritmo tan complejo que fuese casi imposible atacarlo por los métodos tradicionales conocidos por el criptoanálisis, pero este hecho también implicaría un gran esfuerzo tanto de cálculo, como de energía o de almacenamiento. También si se ejecutan tantos cálculos en serie el algoritmo se ralentizaría excesivamente. Así pues aunque el criptosistema fuese excesivamente bueno y seguro, su alto coste podría convertir al algoritmo en un criptosistema inviable en la vida real, y todo ello sin tener en cuenta el gran coste monetario que también conllevaría el proceso.

Del mismo modo, se podría crear un algoritmo diferente que implicase un coste muy bajo o prácticamente ínfimo, pero a su vez, es fácilmente entendible que este criptosistema tendría muy baja seguridad frente a todos aquellos ataques que intentasen descifrarlo, debido a su propia trivialidad.

También se debe prestar especial atención a la elección de la contraseña, ya que es muy común elegir contraseñas que sean fácilmente recordables, como por ejemplo, fechas y nombres. De esta manera no se utiliza la seguridad que aporta el algoritmo, ya que en el caso de producirse un ataque normalmente estas son las primeras claves que se utilizan [14]. Por lo que la elección de la contraseña es un factor muy importante para mantener la seguridad y que en ningún caso debe ser descuidado.

Como se puede ver con estas simples suposiciones, la relación entre el **criptoanálisis** y los **criptosistemas** debe ser siempre muy estrecha. Por este motivo, estas disciplinas siempre deben ir de la mano en todos sus avances, ya que con la aparición de cada nuevo algoritmo se debe buscar un cierto equilibrio entre el **criptoanálisis** y el **criptosistema**. Ya que se debe hallar un criptosistema que sea lo suficientemente seguro, pero sin que esa seguridad implique un alto coste computacional.

En la referente a este compromiso, AES es un algoritmo altamente seguro, ya que en su proceso de cifrado se utilizan una serie de cálculos que no implican un gran tiempo de ejecución pero si aportan una gran complejidad a la hora de descifrarlo. Para comprobar de una forma rápida y sencilla este compromiso, basta con conocer el número de combinaciones posibles existentes para cada tamaño de clave:

Tabla 2.2 Número de combinaciones posibles versus tamaño de clave [15]

Tamaño de clave	Posibles combinaciones
AES 128 bits	$3.4x10^{38}$
AES 192 bits	$6.2x10^{57}$
AES 256 bits	$1.1x10^{77}$

Así pues, si AES sufriese un ataque por fuerza bruta, incluso con el menor tamaño de clave (128 bits), el número de combinaciones posibles es tan grande que implementar ese ataque conllevaría un gran gasto de tiempo y energía que en la práctica hace imposible su ejecución. Estos cálculos tienen un coste extremadamente alto hasta para el ordenador más potente fabricado. Como ya se ha comentado el algoritmo que se va a implementar es AES con un tamaño de clave y bloque de 128 bits.

2.2. Algoritmo AES

En esta sección se narra cómo **AES** se convierte en el estándar de cifrado, la base matemática del algoritmo y el funcionamiento del algoritmo en su versión tradicional, es decir, la aceptada por el NIST.

2.2.1. Inicios del algoritmo

En enero de 1997 [16] el **Instituto Nacional de Normas y Tecnología (NIST)** de los Estados Unidos anunció su intención de convocar un concurso para elegir un nuevo algoritmo de cifrado que sería declarado como estándar y que se utilizaría en los próximos años. Además el ganador del certamen sería el sucesor del algoritmo **DES**, que debido a los avances en la tecnología había quedado obsoleto y comenzaba a comprometer su seguridad. El **DES** se había declarado como estándar previamente en el año 1976.

Finalmente este certamen se convocó meses más tarde de manera oficial. Este concurso estaba dirigido tanto a la comunidad científica como a la empresa. Además el **NIST** quería que cualquier persona o institución pudiesen participar en el proceso, ya fuese enviando propuestas o en el análisis de los algoritmos, así pues todo el proceso se desarrolló de manera pública y abierta.

En la convocatoria de dicho certamen se exponían las especificaciones que debían cumplir los algoritmos para poder optar a ser los ganadores. Entre esas especificaciones destacan: las claves de cifrado podrían ser de diferentes longitudes (128, 192 y 256 bits), el algoritmo sería de cifrado simétrico con bloques de un mínimo de 128 bits, estar disponible software y hardware y finalmente ser de dominio público. Así pues, las propuestas llegaron de todo el mundo.

Además **NIST** declaró "*que estaba buscando un cifrado por bloques con la seguridad del triple-DES pero mucho más eficiente*" [16]. En el verano de 1998 con estas premisas los 15 algoritmos elegidos en la primera ronda fueron:

Tabla 2.3 Seleccionados en la primera ronda para la elección del AES [16]

Algoritmo	Autores	Tipo de autor
CAST-256	Entrust	Compañía
Crpton	Future Systems	Compañía
DEAL	Outerbridge, Knudsen	Investigadores
DFC	ENS-CNRS	Investigadores
E2	NTT	Compañía
Frog	TecApro	Compañía
HPC	Schroeppel	Investigador
LOK197	Lawrie Brown, Josef Pieprzyk, Jennifer Seberry	Investigadores
Magenta	Deutsche Telekom	Compañía
Mars	IBM	Compañía
RC6	RSA	Compañía
Rijndael	Daemen y Rijmen	Investigadores
SAFER+	Cylink	Compañía
Serpent	Anderson, Biham y Knudsen	Investigadores
Tworfish	Counterpane	Compañía

Tras esta primera selección el **NIST** invitó a toda la comunidad científica a que evaluase la seguridad, el coste y las características de implementación de todos estos algoritmos. Con

estos resultados en agosto de 1999 se hicieron públicos los cinco finalistas de la selección, estos fueron: MARS, RC6, Rijndael, Serpent y Tworfish.

Se volvió a someter a estos algoritmos a una revisión más exhaustiva aún y finalmente a una votación, de esta votación salió vencedor Rijndael con un total de 86 votos. De este modo el 2 de octubre del 2000 el NIST anunció de manera oficial que el algoritmo **Rijndael** era el ganador y se convertiría en **Advanced Encryption Standard** (AES), es decir, el nuevo algoritmo estándar de cifrado. Así sería usado por los EE.UU pero también por otros muchos países.

El algoritmo **Rijndael** debe su nombre a sus autores, ya que se llamaban John Daemen y Vicent Rijmen, ambos son de origen belga. Estos especialistas en criptografía, hacían investigaciones en este campo desde hace varios años, así pues el algoritmo Rijndael puede considerarse una evolución del algoritmo Square, desarrollado previamente. Si bien es cierto que hasta llegar a convertirse en AES y ser un algoritmo estándar tuvo que sufrir algunas modificaciones propias del proceso de estandarización.

2.2.2. Base matemática del algoritmo

2.2.2.1. Campos de Galois

El principal detalle a destacar en la base matemática del algoritmo, es que AES opera en el **Campo de Galois** o campo finito. El Campo de Galois es muy útil y usado en criptografía. Aunque el objetivo de este trabajo fin de grado no es conocer ampliamente el **Campo de Galois**, si se explicará brevemente parte de su funcionamiento.

AES interpreta cada segmento de 8 bits como elementos de un Campo de Galois GF (2^8). Estos campos se utilizan en criptografía porque permiten cifrar y descifrar en el mismo cuerpo. Eliminado así los errores fruto de redondeos.

De manera general los campos de Galois con módulo p y grado n se representan:

$$GF(p^n) = \{\lambda_0 + \lambda_1 x + \lambda_2 x^2 + \lambda_3 x^3 + \cdots + \lambda_{n-1} x^{n-1}; \lambda_0, \lambda_1, \lambda_2, \lambda_{3,} \dots \lambda_{n-1} \in Z_p\}$$

Del mismo modo p debe ser un número primo. Cada elemento del campo es un resto de módulo $p(x)$, además $p(x)$ es un polinomio irreducible de grado n, así pues no puede ser factorizado en polinomios de grado menor que n.

En el caso del **AES**, al operar en un **Campo de Galois**, GF (2^8), se representa los 8 bits como un polinomio de grado 7, cuyos coeficientes λ_i solo pueden tomar valores binarios, es decir 0 y 1. De este modo los números hexadecimales con los que trabaja AES se convierten así en polinomios [10].

2.2.2.2. Byte y bits en AES

AES trabaja a nivel de byte, es decir, trabaja con conjuntos de 8 bits. Así pues un byte se puede representar como:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0$$

Donde b_7 es el bit más significativo (MSB) y b_0 el bit menos significativo (LSB). Todos estos coeficientes b_i solo pueden tomar dos valores (0 o 1).

Del mismo modo un byte se puede representar usando notación hexadecimal, ya que cada conjunto de 4 bits se puede considerar un carácter. AES opera en hexadecimal, dos caracteres equivalen a 8 bits y a un byte, también se puede pensar como una equivalencia directa entre la

notación hexadecimal y la binaria. A continuación se muestra una pequeña que sirve para ilustrar mejor este concepto:

Tabla 2.4 Equivalencia hexadecimal a binario

Binario	Hexadecimal	Binario	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Además se verá un ejemplo para entender mejor esta equivalencia, si queremos representar un número cualquiera, por ejemplo 2A:

$$2A_{hexadecimal} = 00101010_{binario} = x^5 + x^3 + x$$

2.2.2.3. Operaciones en los Campos

En los Campos de Galois GF (2^8), las operaciones básicas se utilizan de forma diferente, ya que sufren unas pequeñas transformaciones. En este punto se comentan estas transformaciones de una forma simple, para hacerlas fácilmente entendibles.

En este campo, la **suma** se transforma en una X-OR ($\oplus$ u OR-Exclusiva), en la práctica al sumar dos coeficientes de dos polinomios, el resultado es 0 si los coeficientes son iguales y 1 si los coeficientes son distintos. Así pues la suma es una X-OR bit a bit. El proceso de la **resta** es totalmente análogo.

La **multiplicación** se realiza de la manera tradicional, pero luego se utilizan múltiplos del polinomio $x^8 + x^4 + x^3 + x + 1$, para reducir el grado del resultado. Es decir, se restan sucesivos múltiplos de éste polinomio al resultado de la multiplicación, hasta que se obtiene un polinomio cuyo grado más alto sea menor o igual a 7. De este modo el resultado pertenece al campo GF (2^8).

2.2.3. Fundamentos de diseño

Los creadores de este algoritmo de cifrado lo diseñaron con tres premisas claras, que fueron la guía de su diseño:

- Simplicidad en el diseño.
- Resistencia contra todos los ataques conocidos hasta ese momento.
- Velocidad, código compacto y que fuese operativo en varias plataformas distintas.

Una diferencia muy importante del AES comparado con su predecesor DES, es que AES no utiliza una estructura tipo Feistel [17], ya que en este caso los bits sufren transformaciones pero también se permutan en cada vuelta. El principio de diseño del AES es el "Wide Trail" [16], este principio aporta resistencia al algoritmo frente a ataques de tipo diferencial y lineal. Para conseguirlo se utilizan diferentes capas a lo largo de la ejecución del algoritmo, estas capas también se conocen como layer, a continuación se explica brevemente su funcionamiento [10]:

- Capa de **mezcla lineal**: aporta gran difusión a lo largo de las rondas:

- Capa **no lineal**: utiliza de modo paralelo la S-Box para lograr unas óptimas propiedades de no linealidad.
- Capa para la **adicción de la clave**: utiliza una OR-exclusiva entre los bits del correspondiente estado intermedio y la subclave de ronda.

Así mismo, antes de la primera ronda se aplica la capa de adicción de la clave, esto se realiza para evitar que se ataquen las claves con un ataque basado en texto en claro conocido. En la última rinda se utiliza otra capa de mezcla lineal distinta, para intentar que los algoritmos de cifrado y descifrado sean lo más parecidos posibles.

Es cierto que estas técnicas no son nuevas, ya que algunas habían sido utilizadas en diferentes algoritmos, pero el modo en que las utiliza si aporta seguridad y robustez al algoritmo si lo comparamos con su predecesor. Por ejemplo la permutación inicial y final del DES puede ser descubierta sin conocer la clave, este hecho es un fallo en la seguridad del algoritmo.

2.2.4. Funcionamiento del algoritmo

Como se ha analizado en anteriores apartados, **AES** es un sistema de cifrado por bloques simétrico, es simétrico porque utiliza la misma clave tanto para cifrar como para descifrar y de bloques porque cifra varios símbolos con cada clave.

Como ya se ha comentado **AES** tiene diferentes tamaños de clave, de esta manera se obtienen distintas longitudes de clave, existe así AES de 128, 192 y 256 bits. Del mismo modo, en función del número de bits se tienen diferentes números de rondas como se puede ver en la siguiente tabla:

Tabla 2.5 Número de rondas en función del número de bits [18]

Tipo de AES	N_R	N_K
AES-128	10	4
AES-192	12	6
AES-256	14	8

Donde N_R representa el número de rondas necesarias para llegar a cifrar o descifrar el mensaje y N_K muestra el tamaño de la clave, ya que indica el número de columnas que tiene la clave, siendo 4 el número de filas de la clave.

Anteriormente en este trabajo fin de grado, ya se ha explicado que las implementaciones del algoritmo AES se desarrollan en C y CUDA solamente para un tamaño de clave de 128 bits. Por este motivo, a lo largo de este apartado, se explica de un modo más detallado el funcionamiento del algoritmo AES para 128 bits, aunque esta explicación pueda extenderse a otros tamaños de clave fácilmente. Así pues todo lo que se describe a continuación es para una longitud de 128 bits.

Lo primero que hay que conocer para comprender su funcionamiento es que AES trabaja con bloques, estos bloques como ya se ha comentado tienen una longitud de 128 bits. Además estos bloques se pueden dividir en segmentos de 16 bytes cada uno. A su vez este segmento se puede entender como una matriz de 4x4 bytes o elementos. De este modo se tienen 8 bits por componente. Esta matriz se conoce en AES como estado o state. Esta explicación se puede extender a la clave en el caso de AES-128 bits.

A continuación se muestra la figura 2.5 que ilustra cómo se compone el estado y la clave en AES:

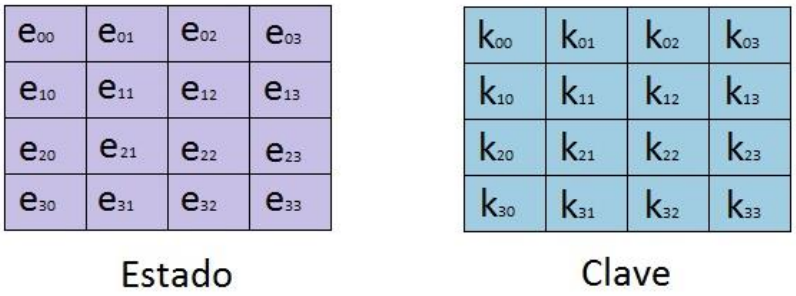

Figura 2.5 Estado y clave en AES-128 bits

2.2.4.1 Modos de operación

Para cifrar un mensaje se divide la totalidad del texto plano en estados, el problema puede surgir cuando el mensaje a cifrar o descifrar no es un múltiplo entero del tamaño del estado. En ese momento aparece la duda para rellenar los huecos que sobran en un estado. Para resolver este problema se pueden utilizar diferentes técnicas, la más simple es completar los huecos con ceros o con cualquier otro estándar, aunque también hay otras técnicas más complicadas, pero realizar un análisis sobre este concepto no es el objetivo del presente trabajo fin de grado.

Los algoritmos de cifrado por bloques presentan diferentes modos de operación. Estos modos se diferencian en la forma que se aplica el algoritmo de cifrado. El modo de operación de las implementaciones de este TFG es **ECB.** A continuación se explica brevemente el funcionamiento de este y otros modos de operación [14]:

- **Electronic Code - Book (ECB):** es el método más sencillo, pero no el más seguro. Consiste en dividir el mensaje en estados, todos los estados se cifran por separado con la misma clave. La principal desventaja es que dos fragmentos de texto plano idénticos se cifran del mismo modo, este hecho puede dar pistas a los criptoanalistas. Una ventaja es que admite cifrar los bloques de un modo independiente, así pues si se produce algún error en el cifrado de los bloques no necesariamente tiene que extenderse en el resto de estados.

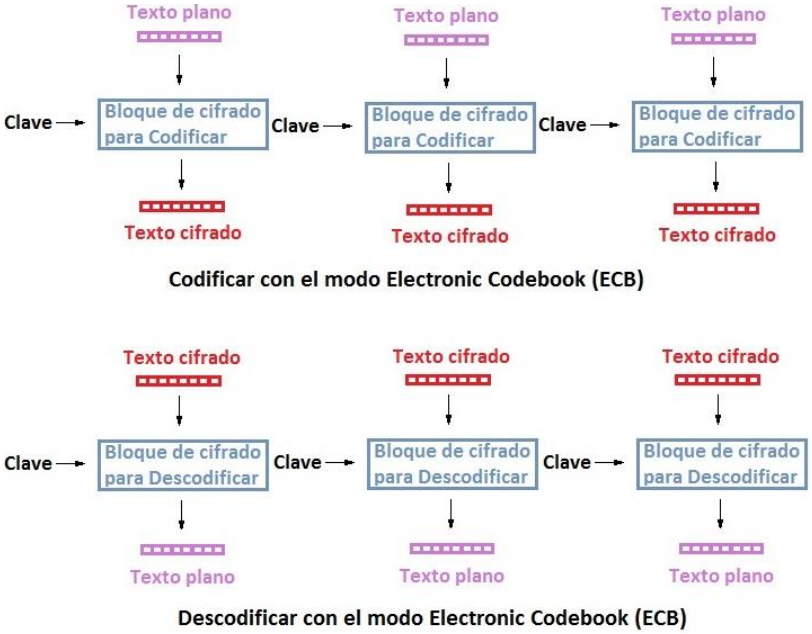

Figura 2.6 Modo ECB

- **Cipher – block Chaining (CBC)**: utiliza un mecanismo de concatenación, es decir, enlaza el siguiente estado a cifrar con el resultado obtenido de encriptar el estado anterior mediante una X-OR. De este modo todos los estados dependen de aquellos que han sido cifrados previamente. Con este modo dos mensajes idénticos se cifran del mismo modo, para evitar que esto suceda se suele añadir un vector de inicialización.

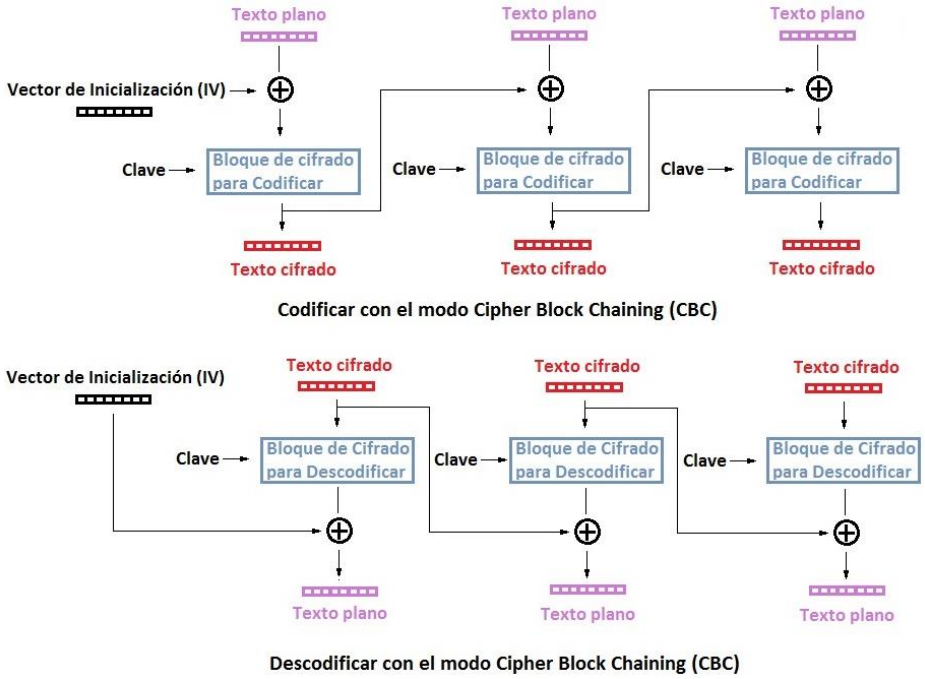

Figura 2.7 Modo CBC

- **Output Feedback (OFB)**: convierte la manera de cifrar en un cifrador de flujo. Utiliza un vector de inicialización que es codificado, este vector se suma al primer estado a codificar con una X-OR. Para codificar el resto del mensaje se utiliza la salida del cifrador del bloque anterior.

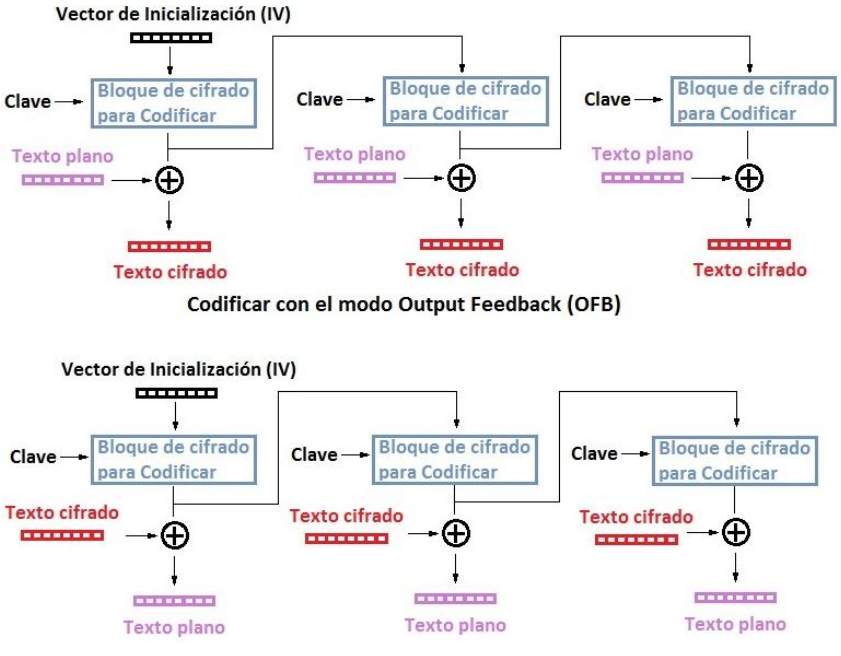

Figura 2.8 Modo OFB

- **Cipher FeedBack (CFB)**: es una especie de cifrador de flujo, funciona de un modo similar al OFB y también utiliza un vector de inicialización. Aunque para realizar la codificación del siguiente estado utiliza el resultado de la X-OR de la vuelta anterior.

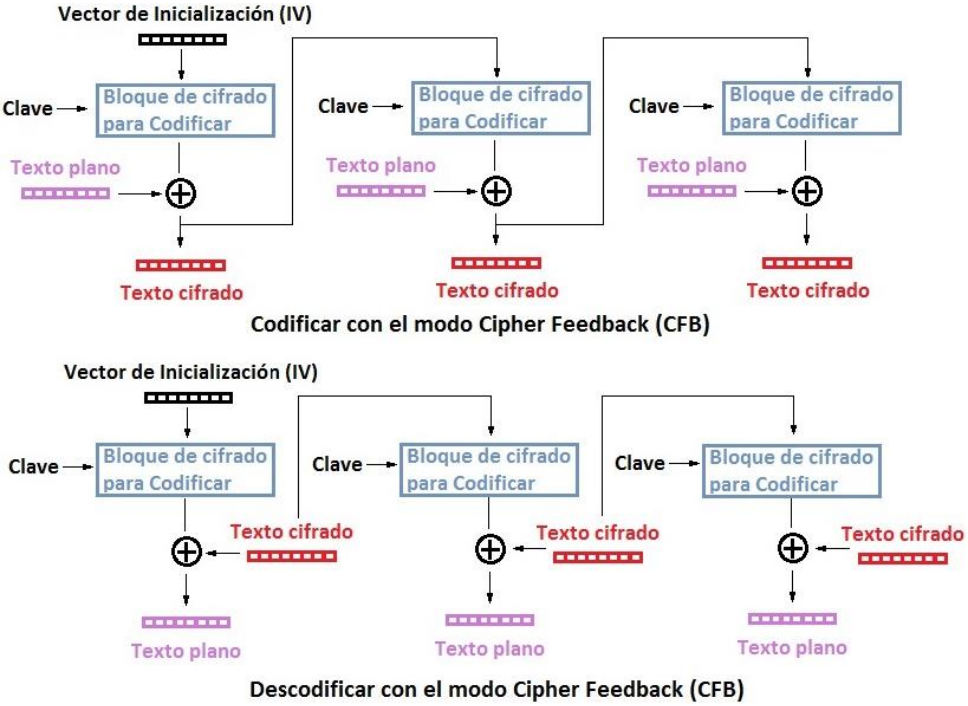

Figura 2.9 Modo CFB

- **Counter (CTR)**: al igual que los dos últimos realiza un cifrado de flujo. En esta ocasión utiliza valores sucesivos de un contador para cifrar. El contador también puede ser cualquier función sencilla. Además realiza una X-OR de un modo similar a OFB Y CFB.

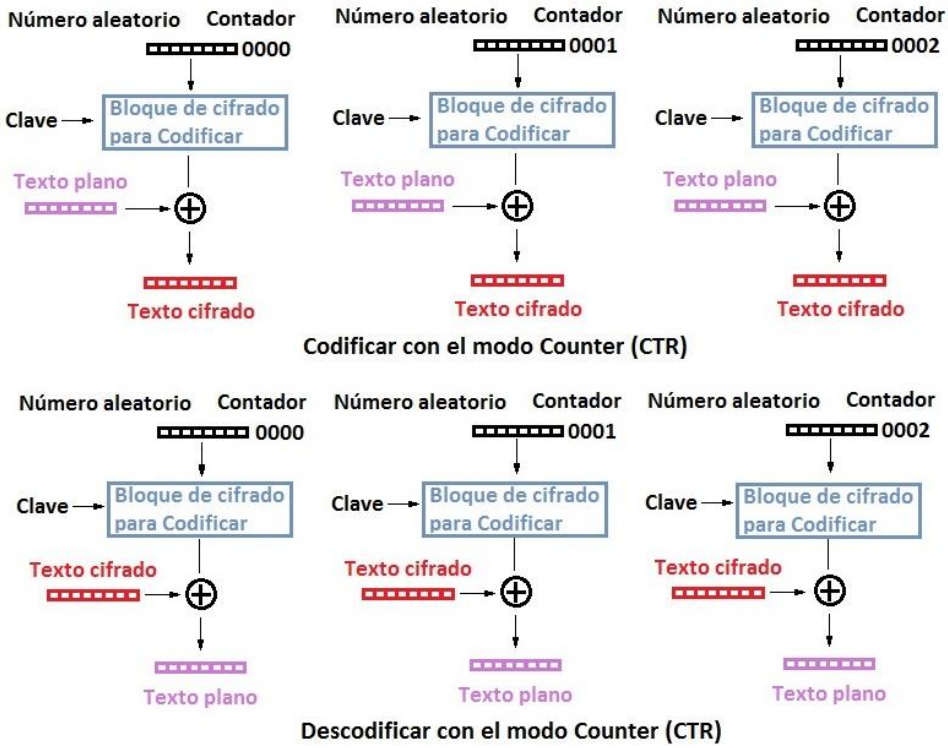

Figura 2.10 Modo CTR

2.2.4.2. Proceso de Cifrado en AES

Una vez comentado la distribución del estado y la clave en AES, así como los principales modos, se va a analizar las rondas del proceso de cifrado para AES – 128 bits en modo ECB. Para ver de manera muy visual las rondas y capas que componen este proceso se tiene el siguiente diagrama como ayuda:

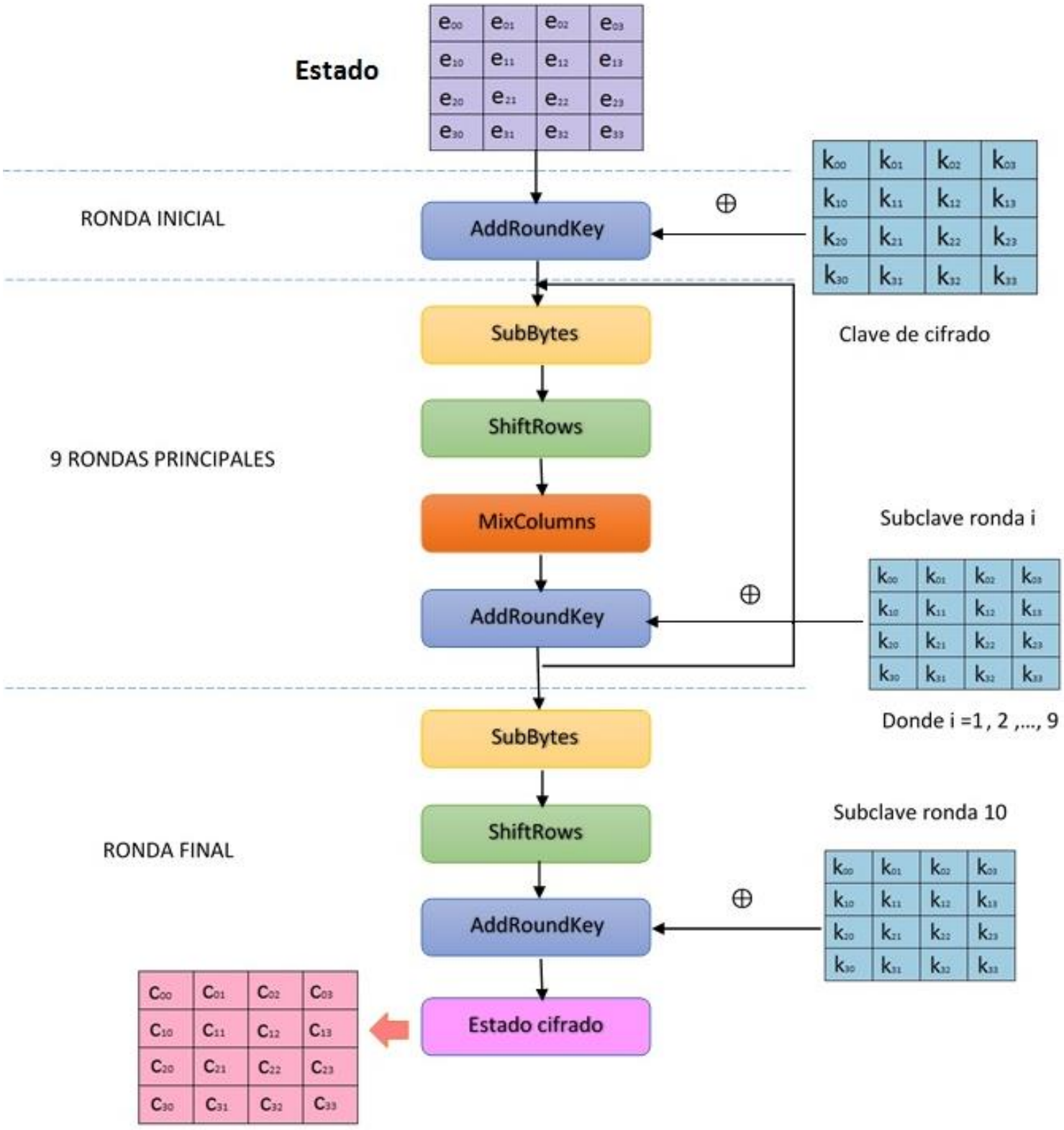

Figura 2.11 Diagrama cifrado AES

2.2.4.3. Proceso de Descifrado en AES

Del mismo modo, a continuación se analizan las rondas del proceso inverso de cifrado, es decir, el proceso para descifrar un estado en el tipo de AES que se está analizando en este TFG:

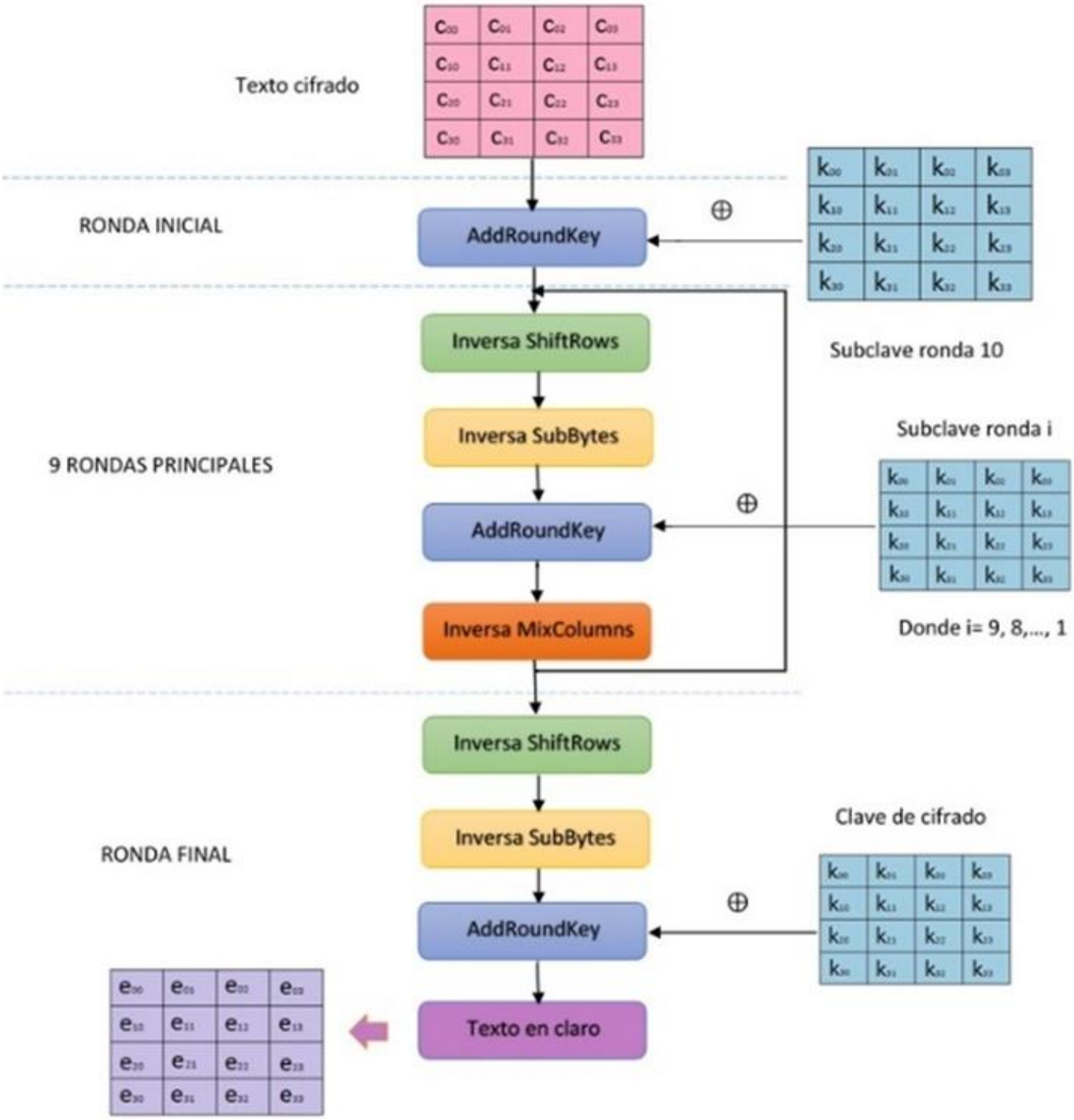

Figura 2.12 Diagrama descifrado AES

Como se puede ver ambos procesos son muy similares, aunque como es lógico el funcionamiento de las rondas inversas difieren a las del proceso directo. De todos modos en el siguiente apartado se detalla el funcionamiento de cada una de las rondas de ambos procesos.

2.2.4.4. Funciones en AES

En el presente apartado se describe de un modo exhaustivo el funcionamiento del algoritmo de cifrado, este hecho hace más fácil la labor de entender correctamente el modo en el cual trabaja este algoritmo.

Como se ve en los dos diagramas anteriores tanto en el proceso de cifrado como en el descifrado se ven tres tipos de rondas. De este modo en ambos procesos aparecen tres tipos de rondas diferentes:

- La **ronda inicial**: se trata de aplicar la operación AddRoundKey entre el estado y la primera clave para el proceso de cifrado y en el proceso inverso entre el texto cifrado y la última clave calculada a partir de la clave de cifrado. En ambos procesos AddRoundKey se utiliza del mismo modo, ya que la X-OR es una operación que se realiza igual tanto en el sentido directo como en el inverso, es decir, la inversa de la operación X-OR es ella misma.
- Las **rondas principales** o **estándar**: se ejecuta en nueve ocasiones, en cada una de esas vueltas se aplican los cuatro tipos de transformaciones. En el proceso de cifrado se aplican las operaciones SubBytes, ShiftRows, MixColumns y AddRoundKey. El proceso de descifrado es totalmente equivalente pero utiliza las funciones inversas, así pues utiliza las transformaciones ShiftRows inversa, SubBytes inversa, AddRoundKey y finalmente MixColumns inversa.
- La **ronda final**: solo se ejecuta una vez y en ella se aplican tres operaciones. En el proceso de cifrado se utilizan SubBytes, ShiftRows y AddRoundKey, mientras que en el proceso inverso se usa ShiftRows inverso, SubBytes inverso y para terminar se aplica un AddRoundKey. En este caso para codificar se utiliza la última clave calculada y para descifrar se utiliza la clave de cifrado con la que originalmente se cifró el mensaje. Se puede comprobar que la ronda final es muy parecida a la ronda estándar, pero en ella no se usa la operación MixColumns.

Además de estas operaciones hay que considerar el cálculo de las claves de ronda, este hecho es equivalente tanto para el cifrado como para el descifrado, ya que el cálculo se realiza del mismo modo en los dos procesos. La única diferencia es el orden en que se usan las claves de ronda ya que se usan justo en el sentido contrario. Para comprender esto más fácilmente se pueden introducir todas las claves calculadas en una matriz, de este modo para cifrar se utilizan las claves en el sentido natural, es decir, la primera clave que se usa son las primeras cuatro columnas, la segunda clave son las siguientes cuatro columnas y así sucesivamente. Mientras que para descifrar se utilizan en sentido contrario, la primera clave que se usa son las últimas cuatro columnas y la siguiente justo las cuatro que se encuentran en la posición anterior.

A continuación se describe el funcionamiento de las funciones que componen el algoritmo, siguiendo las especificaciones del NIST para ello se han utilizado las referencias [16], [19], [20].

Además se debe decir que para que resulte más fácil entender el algoritmo se utiliza el primer vector de test del NIST [21] y su correspondiente clave de cifrado, así pues se utiliza como ejemplo. De este modo el proceso resulta más sencillo de comprender.

Para empezar a desarrollar el funcionamiento del algoritmo, se comienza por explicar cómo se calcular todas las subclaves que son necesarias en las distintas rondas.

2.2.4.4.1. Cálculo de las subclaves

En primer lugar se debe destacar el motivo por el cual se comienza por este punto. Se hace así porque en todas las implementaciones que se desarrollan en este TFG del algoritmo de cifrado, el cálculo de las subclaves siempre se realiza al principio y de manera aislada. Es decir, en el programa en C se calculan todas las subclaves al principio del código, mientras que en el programa en CUDA se realizan al comenzar la parte del código que se ejecuta en la CPU.

A continuación se profundiza más detenidamente en el modo en el que se opera para calcular todas las claves de ronda que se utilizan para cifrar y descifrar. Para ello, se parte de la clave que se ha elegido para cifrar el mensaje.

Se debe comentar que para elaborar completamente el proceso de cifrado son necesarias 11 claves de ronda o subclaves. Así pues para calcular este conjunto de claves se puede utilizar una matriz de 4 x 44 componentes, aunque para utilizar estas claves de un modo más sencillo en el código se introducen estas claves ordenadamente en un vector. Por simplicidad se explica únicamente el cálculo de la primera clave de ronda para el proceso de cifrado en la matriz de claves se coloca en segunda posición, de modo equivalente en el proceso inverso de cifrado esta clave se usa en penúltimo lugar.

Se puede dividir el proceso en dos partes para explicarlo. En primer lugar se calcula la primera columna de la siguiente clave de ronda $k_{i,j}$, (este método se utiliza para las columnas que son múltiplo de cuatro, es decir, la primera columna de cada clave), para ello se utiliza la última columna de la clave anterior $k_{i,j-1}$, en la disposición de matriz que se usa es la columna anterior a la que se está calculando. La primera operación que se utiliza es ***RotWord***, consiste en rotar la columna un byte, es decir, el segundo elemento pasa a ser el primero, el tercero se convierte en el segundo, el cuarto en el tercero y finalmente el primero pasa a convertirse en el último. A este resultado se le aplica la siguiente operación llamada ***SubBytes***, cada elemento es sustituido por el correspondiente de la matriz de sustitución de bytes (S-Box), en el siguiente punto se habla más en profundidad de esta matriz. Finalmente se ejecuta la última instrucción, consiste en aplicar la operación ***X-OR*** entre la primera columna de la clave anterior $k_{i,j-4}$, (cuatro posiciones más delante de la que se está calculando), el resultado de la operación SubBytes y la $columna_n$ de la matriz R_{con} donde n es el número de ronda en la que se usa la clave, en el caso de la primera clave es la primera columna. Entonces se obtiene el resultado de la primera columna de la nueva clave.

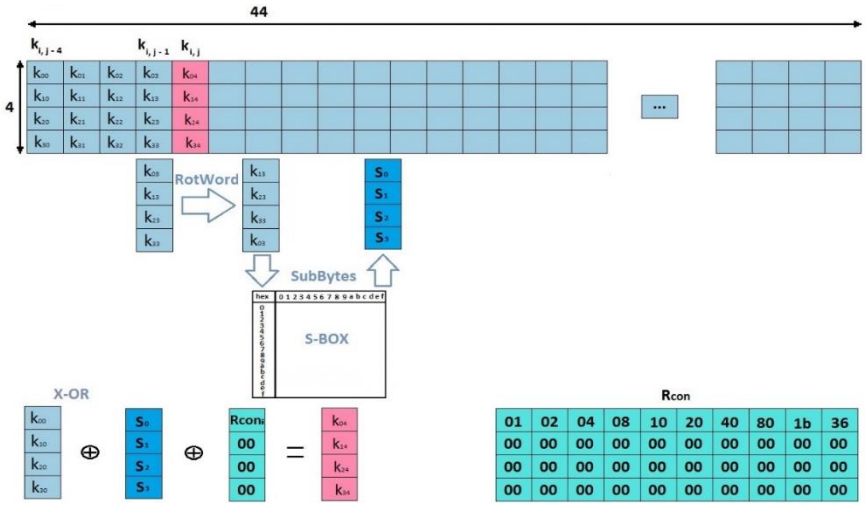

Figura 2.13 Cálculo primera columna de la nueva clave

R_{con} es una matriz que tiene prácticamente todas sus componentes nulas, excepto la primera fila de la matriz. Esta fila está formada por valores constantes, en concreto sus elementos están formados por las potencias de 2, así pues las componentes de la primera fila se pueden expresar de manera general como $2^{(n-1)}$, donde n es el número de rondas. Además al igual que las operaciones se encuentra en el Campo de Galois.

En segundo lugar se realiza el cálculo de las columnas restantes, es decir, las que no son múltiplo de cuatro, se utiliza la operación X-OR. Para calcular la segunda columna de la nueva clave se realiza la operación X-OR entre la segunda columna de la clave anterior y la primera columna de la clave que se está calculando. Esta operación se puede expresar de manera general de la siguiente forma $k_{i,j+1} = k_{i,j-3} \oplus k_{i,j}$, donde i expresa el número de fila y va de 0 a 3 y j es el número de columna y puede tomar valores de 0 a 10 (se utiliza está notación para los índices porque es la misma que se usa para programar). Este procedimiento se puede extender al resto de columnas ya que se puede generalizar para todas aquellas columnas que no sean múltiplo de cuatro. Así pues, solamente hay que realizar una operación X-OR entre la columna de la clave anterior que ocupa la misma posición que la que se está calculando y la columna de la nueva clave que se acaba de calcular, es decir, la columna anterior que se acaba de calcular en la operación anterior.

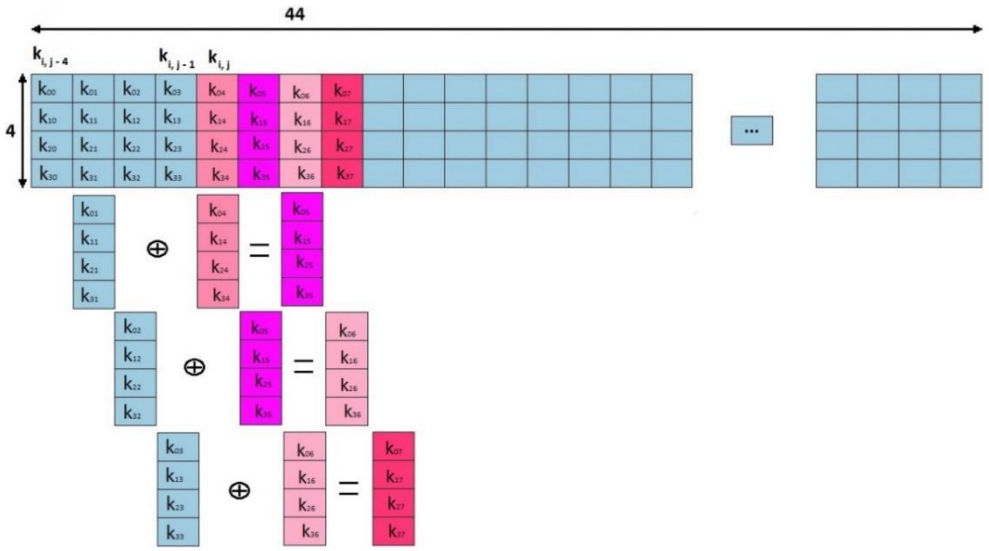

Figura 2.14 Cálculo resto de columnas de la nueva clave

Como se acaba de comprobar el cálculo de todas las subclaves es un procedimiento recursivo y por este motivo no se paraleliza en este TFG. Todas las columnas se calculan a partir de las que se han hallado previamente.

Además para clarificar mejor el cálculo de las subclaves va a hallar la primera clave de ronda de un caso real. Para ello se utiliza como ejemplo la clave de los vectores de test del NIST para 128 bits y modo ECB:

2b	28	ab	09
7e	ae	f7	cf
15	d2	15	4f
16	a6	88	3c

Figura 2.15 Clave de los vectores de test

A continuación se muestra una figura que representa de manera esquemática el cálculo de la primera clave para el ejemplo que se acaba de presentar:

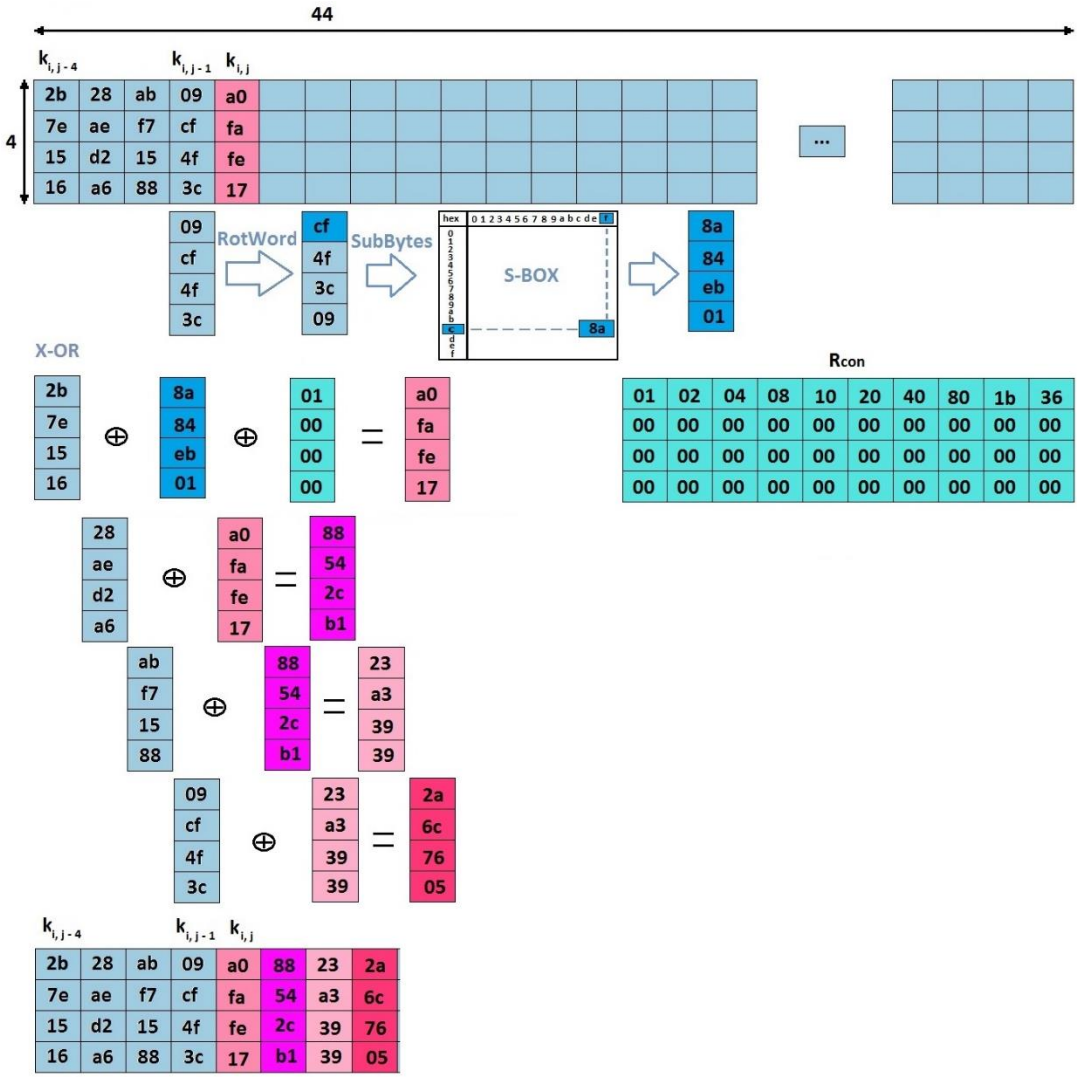

Figura 2.16 Cálculo subclave para la clave de los vectores de test

Se puede ver como el cálculo de las claves utiliza operaciones básicas y sencillas, aunque también puede resultar algo tedioso. Pero hay que prestar especial atención para que todas las operaciones se realicen en el orden correcto, ya que el más mínimo error se puede propagar en todas las operaciones siguientes, debido a la recursividad propia del proceso que ya se ha comentado anteriormente.

Para calcular las claves restantes, que son necesarias para finalizar el proceso, se opera de un modo análogo. De todos modos, se explica someramente como se calcula la siguiente clave y este proceso se puede extender al cálculo de las siguientes claves. Para calcular la segunda clave se utiliza la última columna de la clave anterior y se le aplican las transformaciones **RotWord** y **SubBytes**. Del mismo modo se realiza la operación X-OR con las correspondientes columnas, es decir, la primera columna de la clave anterior, la columna que es el resultado de las transformaciones anteriores y la correspondiente columna de la matriz R_{con}, en este caso la segunda columna. El resto de columnas se calculan del mismo modo que para la clave anterior con la operación X-OR. Así pues se calculan el resto de claves hasta obtener las once claves necesarias.

2.2.4.4.2. AddRoundKey

La función **AddRoundKey** consiste en realizar una operación X-OR entre el resultado de la última transformación aplicada al estado que se quiere cifrar o descifrar y la clave de ronda correspondiente.

En el caso de la ronda inicial la operación se aplica sobre el estado que se quiere cifrar y la clave de cifrado, ya que no se ha realizado ninguna operación previamente. Del mismo modo, para descifrar un mensaje en la ronda inicial se utiliza el texto cifrado recibido y la última clave calculada.

Como ya se ha comentado anteriormente, la función **AddRoundKey** es igual tanto en el proceso de cifrado como en el de descifrado, esto se debe a que la inversa de la operación OR-Exclusiva es ella misma.

En la Figura 2.17 se ve de un modo genérico la operación AddRounKey:

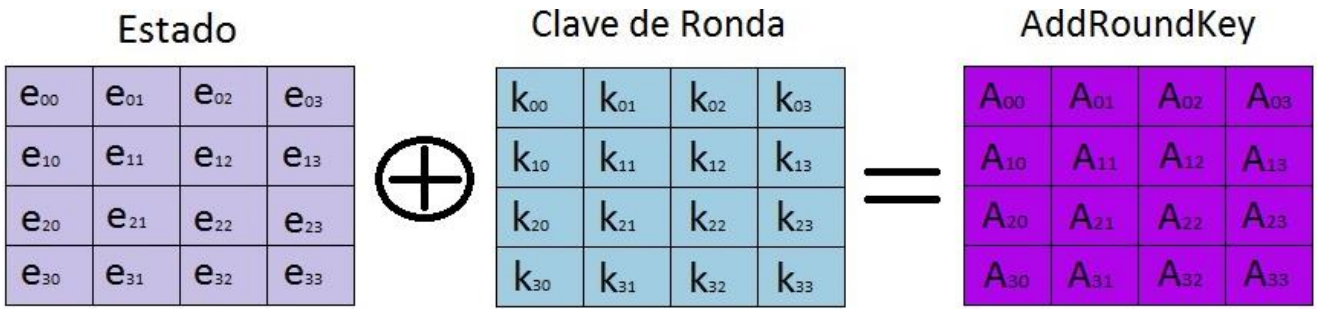

Figura 2.17 AddRoundKey entre el estado y la clave

El resultado de la transformación AddRoundKey puede ser considerado como el estado intermedio sobre el cual se aplica la siguiente transformación, ya que es el estado que se usa en la próxima ronda.

Al igual que para el caso de las claves de ronda se va a utilizar el primer vector de test del NIST, para mostrarlo como ejemplo:

6b	2e	e9	73
c1	40	3d	93
be	9f	7e	17
e2	96	11	2a

Figura 2.18 Vector de test del NIST

Utilizando este vector y la clave de cifrado se obtiene el siguiente AddRoundKey para la ronda inicial:

Figura 2.19 AddRoundKey para la ronda inicial del vector de test del NIST

La única operación que se usa en la función AddRoundKey es la X-OR, esta operación se puede expresar de la forma:

$$AddRoundKey_{i,j} = estado_{i,j} \oplus clave\ ronda_{i,j}$$

Donde $estado_{i,j}$ representa el correspondiente elemento del estado a cifrar/descifrar o el resultado de la transformación anterior, $clave\ ronda_{i,j}$ es el elemento de i,j de la correspondiente clave de ronda y finalmente $AddRoundKey_{i,j}$ es el resultado de la transformación.

El funcionamiento de la operación X-OR se puede considerar elemental o sencilla. Así pues, la complejidad a la hora de cifrar no la aporta la operación, lo hace el cálculo de las claves de ronda. Por este motivo es tan importante el cálculo de las claves de ronda, que se ha explicado previamente. De este modo el resultado del algoritmo (mensaje cifrado o descifrado) depende de la clave que se escoge para cifrar el mensaje.

2.2.4.4.3. SubBytes

La función **SubBytes** consiste en sustituir cada byte del estado por su equivalente en la tabla S-BOX. Esta transformación se trata de una sustitución no lineal que opera de manera independiente sobre todos los bytes.

En la figura adjunta se puede ver de un modo básico el funcionamiento de la función **SubBytes**, pero más adelante se detalla más detenidamente, haciendo especial hincapié en el cálculo de

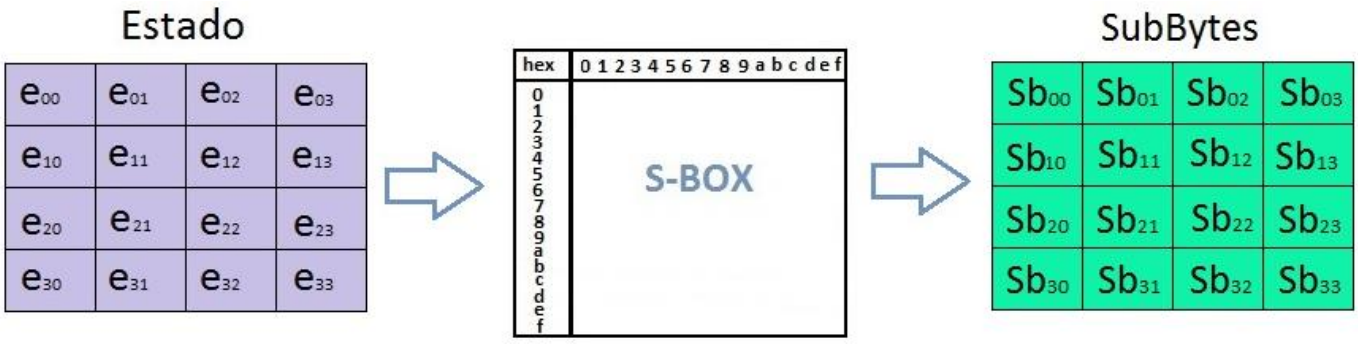

Figura 2.20 SubBytes del estado

la S-BOX.

Esta etapa aporta confusión en el proceso de cifrado, ya que con esta sustitución se intenta esconder la relación que existe entre el criptosistema y el mensaje, manteniendo así su seguridad.

Esta confusión la aporta la tabla de sustitución de bytes, esta tabla también se conoce comúnmente como S-BOX. La S-BOX es una matriz invertible, para formar esta tabla es necesario aplicar estas dos transformaciones:

- Se deben sustituir los elementos del estado por sus inversos para la multiplicación en el campo finito de Galois de la forma GF (2^8). El funcionamiento de estos campos ya se explicó brevemente en un apartado anterior. De este modo se sustituyen todos los elementos por sus inversos, aunque el caso del elemento {00} es diferente, porque se sustituye por sí mismo, ya que este elemento carece de inverso.

- Además se aplica una transformación afín sobre el campo de Galois. Esta transformación se expresa en forma de matriz del siguiente modo:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

  Donde cada elemento b_i representa el bit i del byte completo que se quiere transformar, del mismo modo el resultado de la transformación SubBytes en cada bit se expresa como b'_i.

El resultado de aplicar ambas transformaciones es una matriz con 256 elementos ya que opera a nivel de byte, en esta disposición de matriz se conoce como **S-BOX**. A su vez esta matriz utiliza notación hexadecimal. El uso de la S-BOX acelera y facilita el proceso de cifrado.

Tabla 2.6 Tabla de sustitución de bytes [19]

		Y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	Ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	Fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	Ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	Cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0d	db
	A	e0	32	3ª	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	B	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	C	Ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	D	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	E	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	F	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

A continuación se calcula la etapa **SubBytes** de la primera ronda estándar del primer vector de test del NIST:

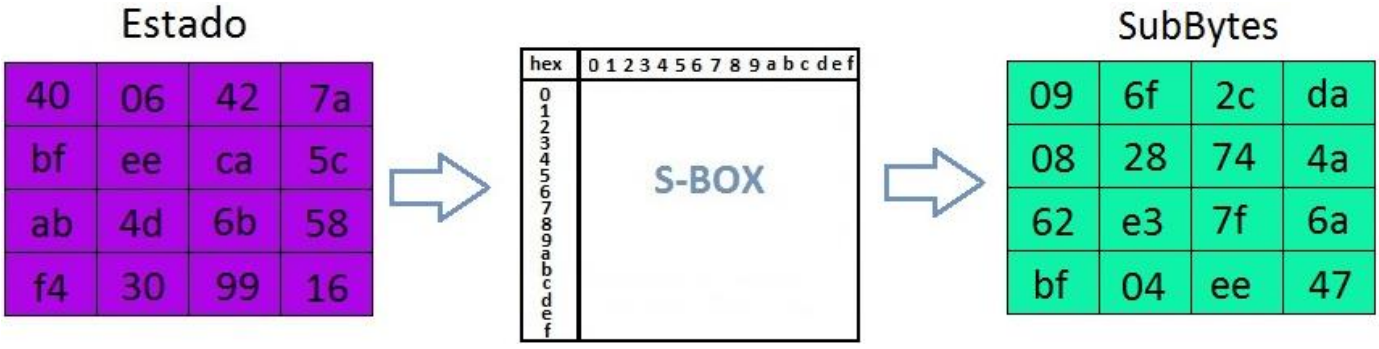

Figura 2.21 Primer SubBytes para el primer vector de test del NIST

Como se puede observar a través de este ejemplo, la implementación de esta función es bastante sencilla gracias al uso de la tabla S-BOX. La S-BOX es una herramienta fácil de usar, ya que simplemente permite sustituir los bytes que compongan el estado por sus equivalentes tras aplicarles las trasformaciones que se han explicado antes.

El funcionamiento de esta tabla es muy cómodo y natural, solamente consiste en dividir cada elemento que compone el estado en dos partes iguales de cuatro bits cada una. De este modo se tiene la parte más significativa (situada más a la izquierda) que representa la fila (x) y los menos significativos (situados más a la derecha) y que representan la columna (y). Así pues con el valor de ambas entradas se obtiene la intersección por la cual se sustituye ese determinado byte.

Para que este concepto quede más claro, se utiliza como ejemplo la primera componente de la salida del primer AddRoundKey del vector de test del NIST que se ha calculado antes (ver figuras 2.21 y 2.20). De este modo se tiene el valor de 40, entonces separamos ambos dígitos, el 4 representa la fila de la S-BOX y el 0 la columna, entonces justo en esa intersección aparece el valor 09, este valor se convierte en la nueva componente del estado que se debe seguir cifrando, se procede del mismo modo con el resto de valores del estado.

2.2.4.4.4. ShiftRows

El fundamento de la función **ShiftRows** son las rotaciones circulares en las filas, es decir, consiste en el deslizamiento de manera cíclica a la izquierda de un determinado número de bytes cada fila del estado.

En esta función cada fila rota un número diferente de bytes, de este modo, la fila 0 no se rota en ninguna posición o no se le aplica transformación alguna, mientras que la fila 1 de desplaza una posición a la izquierda, la fila 2 rota en dos posiciones y finalmente la fila 3 se desplaza en tres posiciones.

En la figura 2.22 se puede ver fácilmente cómo funcionan las rotaciones circulares para un caso general:

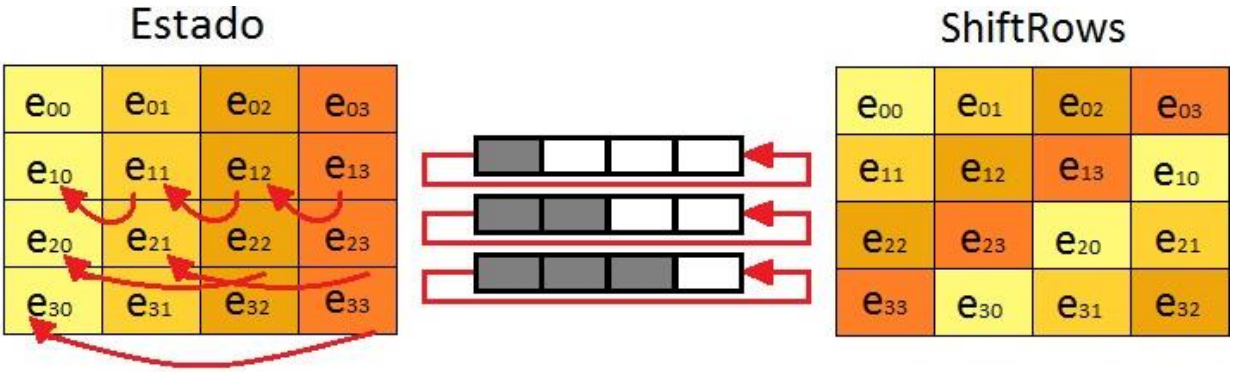

Figura 2.22 ShiftRows del estado

Como se puede comprobar con la anterior figura, en esta transformación solamente se realizan rotaciones circulares hacia la izquierda, ya que en este caso no se aplica ninguna otra operación matemática.

Al igual que en el caso de las funciones explicadas anteriormente, también se elabora el cálculo de la etapa **ShiftRows** para el mismo ejemplo que se hizo antes, es decir, el primer vector de test del NIST, para intentar aclarar mejor la transformación realizada. Dicho proceso se puede ver en la figura 2.23:

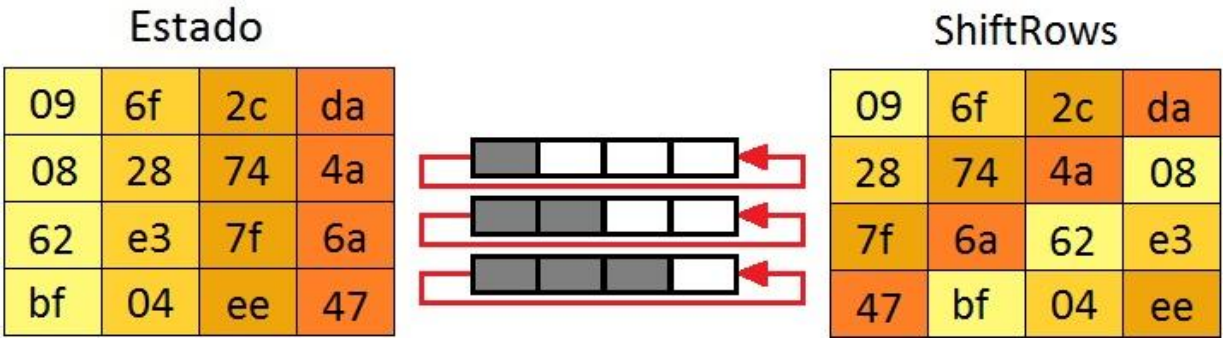

Figura 2.23 Primer ShiftRows para el primer vector de test del NIST

Con este ejemplo, se puede ver cómo se producen las rotaciones circulares de 1, 2 y 3 bytes, además el número de bytes que se rotan coincide con el número de fila en el cual se produce la transformación.

2.2.4.4.5. MixColumns

La última función que falta por explicar y que está involucrada en el proceso de cifrado es la transformación **MixColumns**. La función **MixColumns** actúa sobre los bytes que forman una misma columna en la matriz de estado. Estas columnas son consideradas polinomios sobre el Campo de Galois GF (2^8) y son multiplicadas por el módulo $x^4 + 1$ con el polinomio fijo dado por $MixColumns(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. Así pues, esta multiplicación se puede escribir como $estado'(x) = MixColumns(x) \otimes estado(x)$.

Esta función también se puede expresar de manera matricial como se ve a continuación:

$$\begin{bmatrix} e'_{0i} \\ e'_{1i} \\ e'_{2i} \\ e'_{3i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} e_{0i} \\ e_{1i} \\ e_{2i} \\ e_{3i} \end{bmatrix}$$

Donde i representa el número de columna que se está calculando con la transformación, e_{ji} es la componente del estado de entrada en la transformación y e'_{ji} es la componente del estado de salida.

En la práctica esta función consiste en multiplicar cada una de las columnas que componen la matriz de estado, por la matriz que se acaba de mencionar en la fórmula anterior. Además al operar en el Campo de Galois esta multiplicación sufre las particularidades de estos campos que se comentaron anteriormente en este TFG.

En la figura 2.24 se puede observar el funcionamiento de esta función:

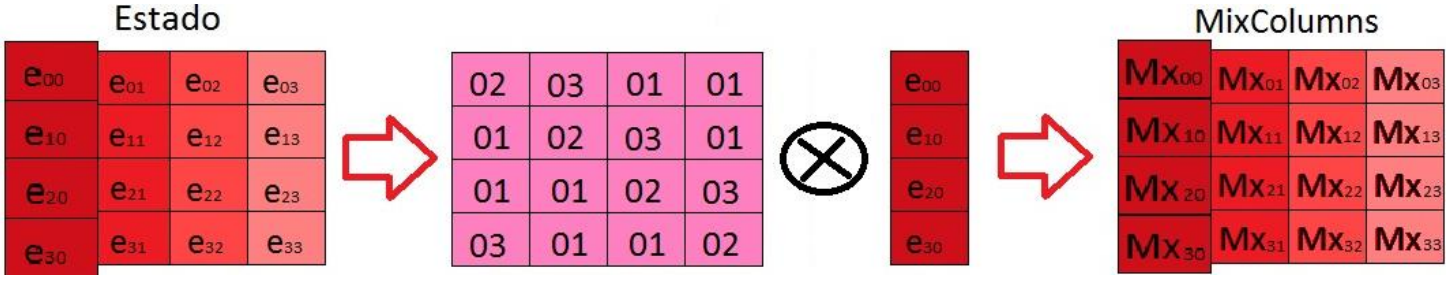

Figura 2.24 MixColumns del estado

A continuación, se calcula la ronda MixColumns para el caso del primer vector de test del NIST, se utiliza este ejemplo del mismo modo que se hizo para las demás funciones en los otros apartados.

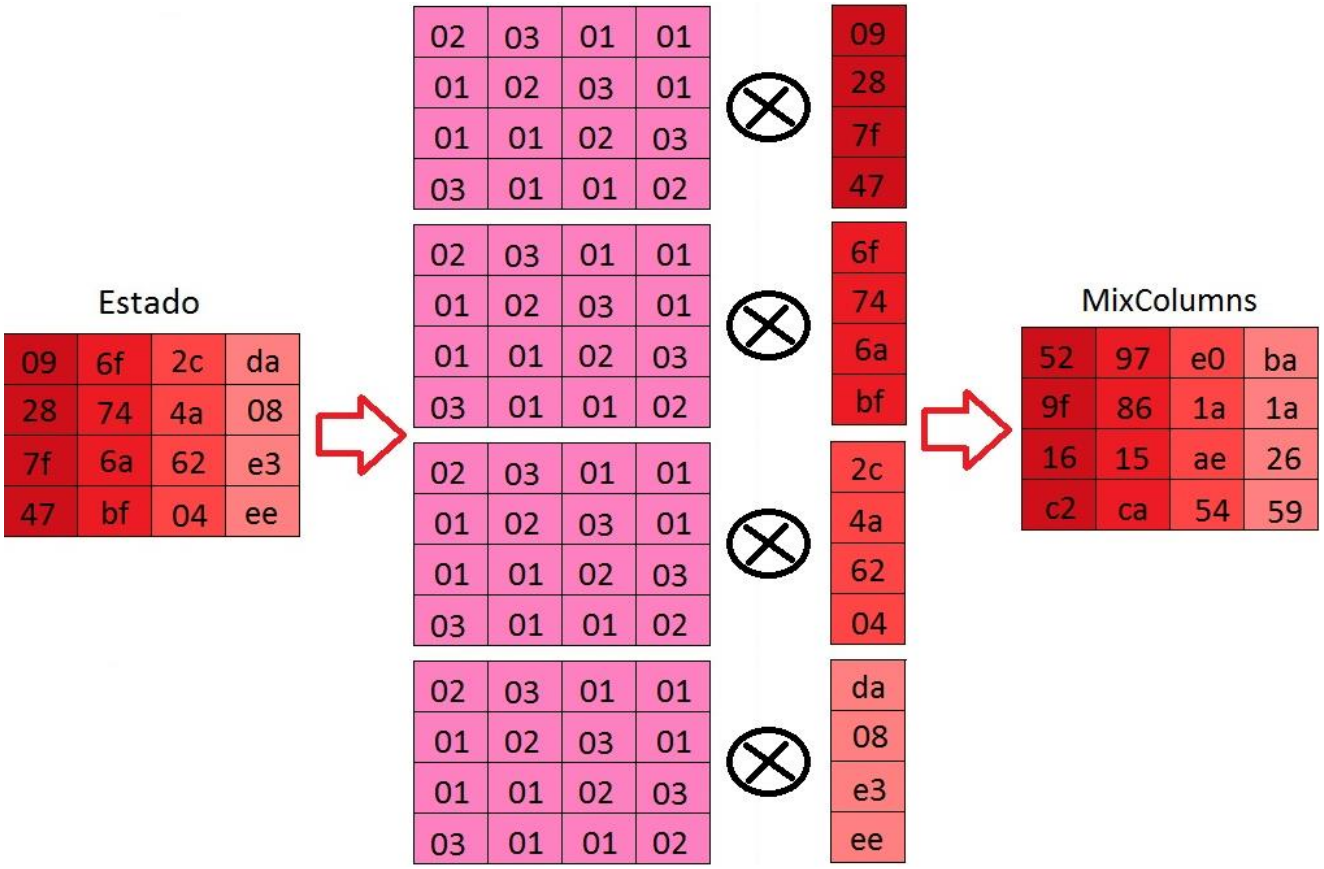

Figura 2.25 Primer MixColumns para el primer vector de test del NIST

Con este ejemplo se ve claramente el funcionamiento de esta función, a través de la multiplicación de las columnas del estado.

2.2.4.4.6. InvSubBytes

Las tres funciones que faltan por describir, son las que se usan en el proceso inverso de cifrado o también llamado proceso de descifrado. Como ya se ha comentado, la función **AddRoundKey** trabaja del mismo modo tanto en el proceso directo como en el inverso, así pues no se vuelve a explicar esta función. Además como el funcionamiento de las funciones inversas es muy similar al de las funciones del proceso directo, su funcionamiento no se explica tan detenidamente como en el caso de las funciones explicadas anteriormente, por este mismo motivo, tampoco se utiliza un ejemplo numérico en cada función para explicar su funcionamiento.

La función **InvSubBytes** también se puede conocer como la inversa de SubBytes y es el equivalente en el proceso inverso de SubBytes. Así pues, su funcionamiento es también muy similar, en este caso sustituye cada elemento del estado por su correspondiente en la S-BOX Inversa.

En la siguiente tabla se puede observar la matriz S-BOX Inversa, al igual que en el caso de la tabla S-BOX para la función SubBytes, los elementos que se representan en esta tabla están en hexadecimal.

Tabla 2.7 S-BOX Inversa [19]

		Y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	D	e	f
x	0	52	09	6ª	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	Ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3ª	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	A	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	B	Fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	C	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	D	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	E	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	F	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

En la figura 2.26 se calcula de un modo genérico la etapa InvSubBytes:

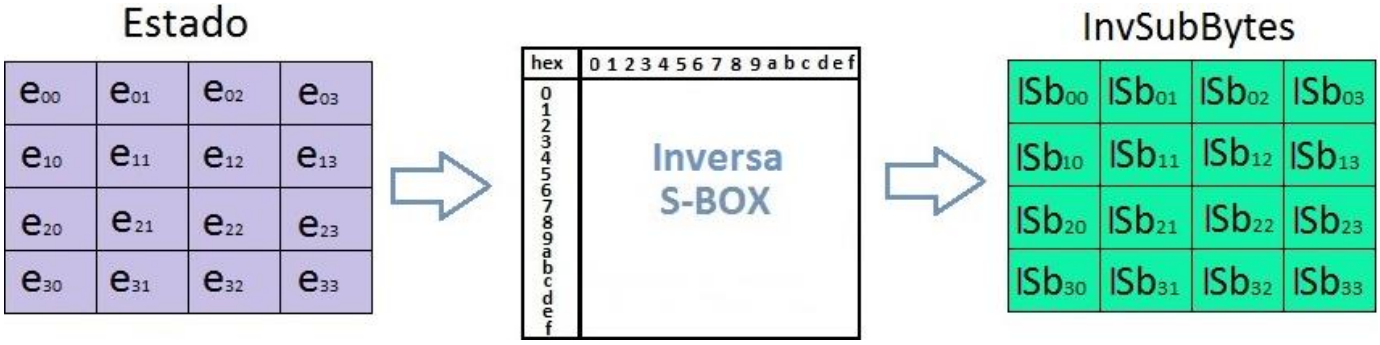

Figura 2.26 InvSubBytes del estado

El funcionamiento de la inversa de S-BOX es totalmente análogo al de S-BOX. Se divide en dos partes cada componente del estado, de este modo se consigue la cifra que designa la fila y la de la columna, entonces en la intersección de ambos aparece el elemento del estado que ha sufrido la transformación InvSubBytes. Además gracias a esta tabla también se acelera y facilita el proceso de descifrado.

2.2.4.4.7. InvShiftRows

La función **InvShiftRows** es la equivalente a la función ShiftRows, su funcionamiento es muy similar ya que también utiliza las rotaciones circulares, pero en este caso los deslizamientos se producen hacia la derecha. Además al igual que en el caso de la función análoga, cada fila rota un número distinto de bytes, es decir, la fila 0 no rota, la fila 1 se desliza un byte a la derecha, la fila 2 rota dos bytes y finalmente la fila 3 rota tres bytes.

Así pues, se podría decir que la función **InvShiftRows** tiene justo el funcionamiento contrario al de ShiftRows. Debido al funcionamiento de esta función este hecho es fácilmente entendible, ya que el uso de las rotaciones circulares es una propiedad muy visual.

En la figura 2.27 se puede ver las rotaciones circulares y su uso en la función **InvShiftRows** para un caso genérico.

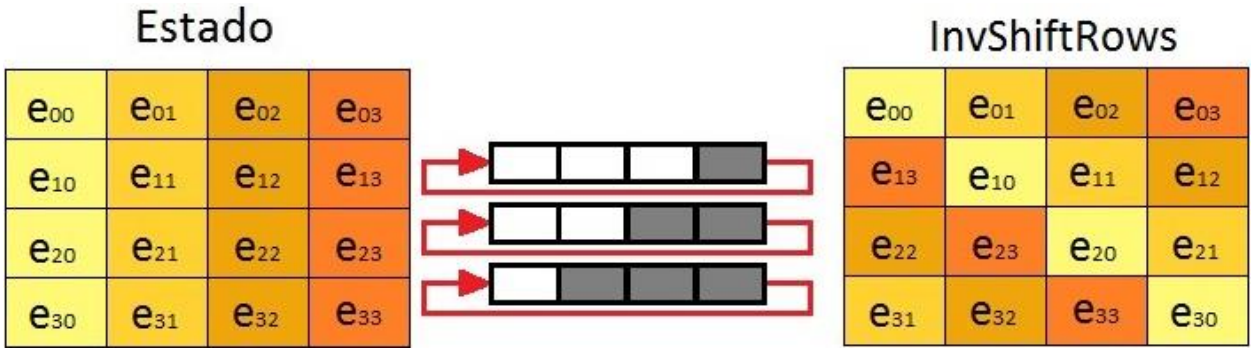

Figura 2.27 InvShiftRows del estado

Se observan las rotaciones de 0, 1, 2 y 3 bytes a la derecha en las filas 0, 1, 2 y 3 respectivamente. Como se puede ver las rotaciones son justo las opuestas a las que sufría el estado durante la función ShiftRows. Esta condición es totalmente necesaria para poder descodificar el mensaje correctamente.

2.2.4.4.8. InvMixColumns

La función **InvMixColumns** por ser la inversa de MixColumns, tiene un funcionamiento muy similar al de esta transformación. Al igual que en el caso de esa transformación, esta función actúa sobre las columnas que forman el estado.

Aunque en este caso, la matriz por las que se multiplican esas columnas es diferente a la de la función MixColumns. Esto se debe a que la función **InvMixColumns** es la opuesta. A continuación se explica cómo se calcula esta matriz de cambio necesaria para esta transformación.

Las columnas del estado en **InvMixColumns** son tratadas como polinomios sobre el Campo de Galois GF (2^8) y son multiplicadas por el módulo $x^4 + 1$ con el polinomio fijo dado por $MixColumns^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$. Esta multiplicación se puede escribir como $estado'(x) = MixColumns^{-1}(x) \otimes estado(x)$.

Además la operación InvMixColumns se puede expresar de manera matricial, de un modo similar al caso MixColumns:

$$\begin{bmatrix} e'_{0i} \\ e'_{1i} \\ e'_{2i} \\ e'_{3i} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \cdot \begin{bmatrix} e_{0i} \\ e_{1i} \\ e_{2i} \\ e_{3i} \end{bmatrix}$$

Al igual que en el caso de MixColumns i índica el número de columna que se está calculando con la función, e_{ji} es la componente del estado de entrada en la transformación y e'_{ji} es la correspondiente componente del estado de salida.

Así pues, el funcionamiento de esta función es muy similar al de MixColumns, ya que su funcionamiento básico es multiplicar una a una las columnas que componen el estado por una matriz, pero como es lógico esta matriz es distinta a la del caso directo, aunque entre ellas exista una relación.

A continuación, se muestra la figura 2.28 para comprender mejor el funcionamiento de la función **InvMixColumns**:

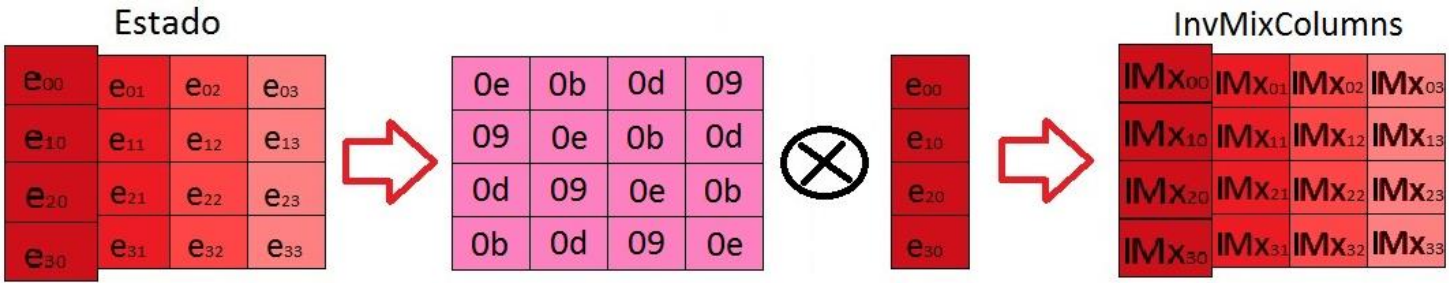

Figura 2.28 InvMixColumns del estado

El resto de columnas del estado se calculan de la misma manera, solo hay que multiplicar cada columna por la matriz anterior, de este modo se obtienen cada una de las columnas que forman el nuevo estado.

Capítulo 3. Entorno legislativo

3.1. Introducción

Como ya se ha comentado al inicio del presente TFG, la **seguridad informática** es un tema tan actual como importante en el mundo de las telecomunicaciones en el cual se mueve la sociedad actual. Ya que, por ejemplo, es muy común realizar operaciones sensibles como el envío de correos electrónicos o compras de manera online, estas actividades se pueden realizar de un modo cómodo y sencillo, pero normalmente nunca nos paramos a pensar del riesgo que estos movimientos pueden poseer. Además de estas actividades cotidianas, también existen otras esenciales en las que se debe preservar especialmente la privacidad, algunas de estas acciones son las llamadas telefónicas y los datos bancarios, pero también hay que tener un especial cuidado con los datos gubernamentales, ya que requieren una seguridad al más alto nivel.

Por este motivo, es cada vez más habitual que la sociedad intente buscar soluciones para mantener su seguridad y privacidad. Así pues, de una manera constante surgen habitualmente nuevos procedimientos y medidas que intentan solucionar esta problemática de un modo cotidiano. De la misma manera, es necesario que esta evolución sea constante a lo largo del tiempo, porque si no se corre el riesgo de que los procedimientos usados queden obsoletos y anticuados.

Con este propósito aparecen constantemente algoritmos de cifrado que son utilizados para cifrar mensajes y mantener la privacidad de toda la sociedad. Uno de estos algoritmos de cifrado es el **AES**, que como ya se ha dicho anteriormente es el algoritmo considerado como estándar para este tipo de procedimientos, además es el algoritmo que se desarrolla y analiza en este TFG.

Actualmente, existen multitud de organismos tanto institucionales como empresariales que se dedican a intentar resolver la problemática de la seguridad. Todas estas entidades se dedican principalmente a buscar nuevos métodos y medios, que permitan mantener la seguridad e innovar con nuevas metodologías, pero también se dedican a probar la seguridad de las soluciones ya existentes.

A lo largo de este capítulo se enumeran brevemente algunas de las leyes que deben regular la seguridad en este tipo de comunicaciones y entornos. Además también se comentan y describen algunos de los organismos que deben estar presentes en la toma de este tipo de decisiones

3.2. Marco regulador

Actualmente existen numerosas leyes que se encargan de regular la seguridad en algunas de estas transacciones, así como a intentar proteger los datos de carácter privado de toda la población. Estas leyes tienen diferentes ámbitos de actuación ya que es un problema global, de esta manera en nuestro caso existen leyes españolas pero también normativas europeas a este respecto. Se puede decir que todos los países tienen leyes y organismos relacionados con la seguridad informática, aunque a priori pueda parecer que solo las grandes potencias mundiales dispongan de ellas.

A este respecto en España, posiblemente las leyes más importantes y conocidas sean [22] la **Ley Orgánica de Protección de Datos de Carácter Personal** (LOPD), la **Ley General de las Telecomunicaciones**, la **Ley Servicios de la Sociedad de la Información y Comercio Electrónico** (LSSI-CE) e incluso la **Ley de Propiedad Intelectual** (LPI).

La **LOPD** es la Ley Orgánica 15/1999 del promulgada el 13 de diciembre 1999 [23] su principal objetivo es proteger y garantizar el tratamiento de los datos personales, así como garantizar de un modo correcto la gestión de ficheros automatizados.

La Ley 32/2003 del 3 de noviembre es la **Ley General de las Telecomunicaciones** [24], esta ley ha sido modificada en varias ocasiones y regula el marco en el cual se mueve el mundo de las comunicaciones electrónicas y la conservación de datos entre otros conceptos.

La **LSSI-CE** es la Ley 34/2002 del 11 de julio de 2002 [25] que regula entre otras muchas cosas las comunicaciones electrónicas por vía electrónica, los contratos electrónicos y la transmisión de contenidos por las redes de telecomunicaciones.

La **LPI** es el Real Decreto Legislativo 1/1996 de 12 de abril de 1996 [26], en la cual se regula los derechos de autor de las obras literarias, artísticas y científicas, así como su divulgación y publicación.

Aunque también existen otras leyes que operan en este ámbito como la ley 59/2003, del 19 de diciembre de Firma Electrónica; el Real Decreto 3/2010 del 8 de enero que regula el Esquema Nacional de Seguridad en el ámbito de la Administración Electrónica y el Real Decreto 4/2010, del 8 de enero, por el que se regula el Esquema Nacional de Interoperabilidad en el ámbito de la Administración Electrónica [22].

De esta manera el algoritmo AES puede ser utilizado por ejemplo para cifrar información personal, archivos o el contenido de las bases de datos. Así pues, gracias a este algoritmo se pueden cumplir algunas de estas leyes que buscan la seguridad en la red o en los datos almacenados. Aunque también se debe comentar que el algoritmo AES no es el único procedimiento para cifrar, pero si es muy utilizado ya que actualmente es el estándar.

Además existen numerosas instituciones encargadas de estudiar y regular estos mecanismos relacionados con la seguridad, debido a la gran importancia de este tema. Estos organismos pueden ser de toda clase, ya que se encargan de estudiar estas medidas tanto las universidades, como los investigadores y toda clase de gobiernos, ya sea para procedimientos civiles como militares o gubernamentales, relacionados con la inteligencia y el espionaje.

Quizás el ejemplo más inmediato de este tipo de organizaciones sean las Agencias de Seguridad Nacional de todos los países, pero también existen otros organismos civiles encargados de regular estas acciones a lo largo de todo el mundo. De esta manera trabajan en este tipo de materias en EE.UU tanto el **NIST** como la **NSA**.

Del mismo modo, en España existen este tipo de organismos regulados por diferentes Ministerios, por ejemplo, este es el caso del Instituto Nacional de Tecnologías de la Comunicación o **INTECO**. INTECO está regulado por el Ministerio de Industria, Energía y Turismo, y se encarga de desarrollar y analizar técnicas para intentar preservar la ciberseguridad y la protección de la privacidad. Además numerosas universidades españolas hacen investigaciones en este ámbito. Pero también existen otras muchas entidades interesadas en la criptografía y la seguridad desde otro punto de vista, este es el caso del CCN o incluso aunque en una escala diferente el CNI.

Ya que aunque en un principio pueda no parecerlo, la criptografía y la seguridad informática es algo que implica a todas las personal de manera cotidiana, pero es también un concepto esencial en materia de espionaje al más alto nivel.

3.3. Aplicaciones AES

En este apartado se comentan algunas de las aplicaciones de AES, aunque algunas ya se han comentado brevemente en el punto anterior. Así pues, AES puede ser utilizado, por ejemplo, para cifrar documentos que contengan información sensible, el procedimiento es muy sencillo, basta con elegir una contraseña y sustituir cada carácter del documento por su equivalente en el código ASCII en hexadecimal. Después se rellenan diferentes estados con esos caracteres en hexadecimal y se procede a cifrar el documento. De un modo análogo se puede proceder a encriptar bases de datos o incluso el contenido de un disco duro. También se podría usar este procedimiento para codificar contraseñas y otro tipo de archivos.

Aunque existen multitud de diferentes sistemas de cifrado, AES es ampliamente utilizado ya que es el estándar actual, además su ejecución es relativamente fácil y es bastante seguro. Sin embargo, AES tiene diferentes modos de operación y el modo ECB (que es el que se implementa en este TFG) es el que aporta una menor seguridad. Así pues, para conseguir una seguridad óptima o a un nivel más alto se debe utilizar otro de los modos de operación detallados anteriormente.

AES también puede utilizarse en alguno de los protocolos **RFC**, estos protocolos son publicados por el IETF con el objeto de normalizar y regular el uso de la información y el transporte de datos en Internet.

Además AES puede usarse en la seguridad de redes inalámbricas tipo WLAN, ya que es utilizado dentro del protocolo **IEEE 802.11**, IEEE es otra entidad sin ánimo de lucro que busca promover la estandarización. IEEE 802.11 detalla el modo en que dos o más nodos inalámbricos se pueden reconocen y transmiten información entre ellos, así pues este procedimiento define el Conjunto Básico de Servicios (BSS) [27]. Existen diferentes variaciones de la norma, en concreto, AES se usa en el estándar 802.11i para reforzar la seguridad en la autentificación de usuarios y mejorar la robustez de los métodos de cifrado [28]. De la misma manera como evolución de este protocolo 802.11i surge el procedimiento WPA2 para proteger redes inalámbricas, su funcionamiento está basado en AES y aparece para resolver los errores y vulnerabilidades de su predecesor.

También existe otro conjunto de protocolos llamado **IPsec**, que regula las comunicaciones cifrando y autentificando todos los paquetes IP que componen un flujo de información. Además **IPsec** contiene protocolos para el establecimiento de claves de cifrado. Concretamente el protocolo RFC 4309 utiliza AES en contador con el modo CBC-MAC (CCM) como mecanismo para proporcionar confidencialidad y autentificar el origen de los datos [29]. La evolución de este tipo de protocolos es constante en el tiempo, ya que aparecen protocolos nuevos en función de las necesidades.

Del mismo modo, también existen un conjunto de instrucciones de codificación denominadas Intel **AES**-**NI**. AES-NI mejora el funcionamiento del algoritmo AES y acelera la codificación de los datos, estas instrucciones son válidas para los procesadores de las familias Intel Xeon e Intel Core. AES-NI está compuesto por siete nuevas instrucciones, que aportan una mayor seguridad y una protección más rápida y asequible de los datos. **AES**-**NI** implementa algunos pasos intensivos en el hardware y mejora la manipulación y generación de las matrices, además de mejorar la multiplicación [30]. De este modo se acelera y fortalece la implementación de AES, lo que mejora también su rendimiento.

Esta puede ser una solución sencilla para proteger datos confidenciales, ya que AES es el estándar más utilizado. Así pues, la tecnología **AES-NI** puede facilitar y acelerar el proceso de cifrado, este hecho permite acercar el proceso de cifrado a cualquier empresa a la vez que la hace cumplir con la normativa vigente en materia de confidencialidad y protección de datos.

3.4. Seguridad AES

Como ya se ha comentado, AES es un algoritmo altamente seguro y distintas organizaciones declaran el algoritmo como suficientemente seguro para su uso en información no clasificada. Incluso algunas dicen que se podría usar en información calificada como TOP SECRET [31].

Anteriormente en este TFG ya se ha hablado sobre los ataques por fuerza bruta y de cómo no resultan una opción viable para el algoritmo AES, ya que el número de posibles combinaciones de claves es tan grande que hacen el tiempo de cómputo excesivamente grande. Así pues, con los ordenadores y procesadores actuales los ataques por fuerza bruta no son una opción factible para AES.

Otro tipo de ataques son los de canal lateral, que compara el consumo de potencia del dispositivo y las operaciones que se realizan en la ejecución del proceso de cifrado. Así pues, se trata de un proceso estadístico que pretende atacar el algoritmo [32]. De la misma manera, también se puede proteger los algoritmos frente a este tipo de ataque enmascarando el texto plano.

Se debe considerar que por tratarse AES el algoritmo estándar, numerosas instituciones, universidades e investigadores trabajan constantemente para intentar romper la seguridad de este algoritmo. Por lo que cada cierto tiempo aparecen nuevos estudios que aunque no consigan romper completamente su seguridad si pueden facilitar este proceso. De esta manera surgen múltiples investigaciones que, al menos en la teoría, significan un paso adelante en este asunto. Por ejemplo, una de esas investigaciones es la de Andrey Bogdanov, Dmitry Khovratovich y Christian Rechberger que desarrollaron un ataque que permitía obtener la clave de cifrado cuatro veces más rápido de lo estimado [33]. Aunque este avance no compromete la seguridad de AES, ya que el número de combinaciones posibles sigue siendo excesivamente grande para romper el algoritmo.

Estos son solo algunos de los ataques que se pueden implementar, pero con ellos se puede ver como por el momento no se han obtenido resultados concluyentes que comprometan la seguridad de AES.

Aunque también se debe tener en cuenta que las investigaciones sobre AES están en una evolución constante, por lo tanto no se sabe si en algún momento se podrá romper la seguridad del algoritmo. Pero en la actualidad se puede decir que AES es un algoritmo altamente seguro, ya que su seguridad no ha sido comprometida de un modo viable en ningún estudio y tampoco parece estar próxima por el momento.

Capítulo 4. Entorno de trabajo

4.1. Introducción

Todos los resultados obtenidos, a través de las diferentes implementaciones que se han elaborado durante la ejecución de este TFG, se han realizado en el mismo entorno de trabajo. De esta manera se pueden comparar correctamente todos los resultados obtenidos durante las ejecuciones de las implementaciones, así como sus características y propiedades más importantes. Ya que si los datos fuesen tomados en diferentes entornos con características distintas, los datos obtenidos no serían extrapolables a otros entornos y por lo tanto no sería correcto compararlos.

Así pues en el presente capítulo se describen brevemente las características primordiales de los dos principales elementos que pueden limitar los resultados obtenidos con los dos tipos de implementaciones desarrolladas. Estos dos elementos limitantes son el ordenador del laboratorio donde se desarrolla este trabajo y la tarjeta gráfica (**GPU**) que se usa para paralelizar el código.

De la misma manera en esta sección también se detallan algunas de las características de la plataforma utilizada para realizar este estudio. Esta herramienta básica de desarrollo es Visual Studio.

4.2. El PC

El ordenador utilizado para realizar este TFG es propiedad de la Universidad Carlos III y está situado en uno de los laboratorios pertenecientes al departamento de Tecnología Electrónica. Las principales características de este ordenador se describen con detalle en la siguiente tabla adjunta:

Tabla 4.1 Características del PC

Procesador	Intel® Core™2 Duo CPU E6750 @ 2.66GHz
Memoria RAM	4.00 GB
Tipo de sistema	Sistema operativo de 64 bits
Entorno operativo	Windows 7

Como se puede ver, de una manera rápida y fácil, observando simplemente estas características, no se trata de un ordenador excesivamente moderno. Pero también se debe decir que estas características son más que suficientes para programar, ya que ese es el principal objetivo a desarrollar en este TFG.

4.3. La Tarjeta Gráfica

La tarjeta gráfica utilizada es de la marca **Nvidia**, concretamente esta tarjeta es del modelo es **GeForce GTX 550 Ti**. Cada modelo de tarjeta presenta variaciones en unas características determinadas. Estas especificaciones pertenecientes a cada tarjeta, se pueden consultar en la correspondiente página web de Nvidia. Otra manera de conocer esas especificaciones, es consultando los ficheros que se generan automáticamente al instalar la correspondiente herramienta de desarrollo, en este caso Visual Studio, estos ficheros se generan al instalar CUDA y la herramienta de desarrollo.

Concretamente, en el caso de la tarjeta utilizada, es decir, Nvidia GeForce GTX 550 Ti, se muestran las principales características en la siguiente captura de pantalla del fichero generado automáticamente:

```
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 550 Ti"
  CUDA Driver Version / Runtime Version          6.0 / 6.0
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073741824 bytes)
  ( 4) Multiprocessors, ( 48) CUDA Cores/MP:     192 CUDA Cores
  GPU Clock rate:                                1800 MHz (1.80 GHz)
  Memory Clock rate:                             2052 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 393216 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 6.0, NumDevs = 1, Device0 = GeForce GTX 550 Ti
Result = PASS
```

Figura 4.1 Captura del fichero con las características de la tarjeta

Además de este fichero, también se genera otro automáticamente relativo al ancho de banda. Aparecen diferentes anchos de banda en función los dispositivos en los que se produce la transferencia de datos, es decir, host to device, device to host y finalmente device to device. Los resultados se pueden ver en la siguiente captura:

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GTX 550 Ti
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     2563.0

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     1926.4

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     71819.8

Result = PASS
```

Figura 4.2 Captura del fichero con las características del ancho de banda

La tarjeta gráfica o **GPU**, es el elemento que utiliza la tecnología de Nvidia para paralelizar la ejecución de los algoritmos ejecutables mediante CUDA y la herramienta Visual Studio. Este concepto esencial y el funcionamiento de este procedimiento, se detalla en profundidad en los capítulos posteriores.

4.4. Microsoft Visual Studio

Microsoft Visual Studio es la principal herramienta de desarrollo que se utiliza para elaborar este TFG, aunque se debe decir que también se han utilizado otras que se describen más adelante. Visual Studio es un entorno de desarrollo integrado [34] diseñado especialmente para sistemas Windows.

Visual Studio es una plataforma que permite programar en diferentes lenguajes de programación, por ejemplo se puede programar en Visual Basic, C, C++, C#, Fortran, Java,… En el caso del presente trabajo los lenguajes de programación utilizados son el tradicional lenguaje C y lenguaje C con extensiones en CUDA para paralelizar el código. En este trabajo se programa en esos dos lenguajes para comparar los resultados obtenidos en la ejecución del algoritmo de cifrado AES.

Esta herramienta se usa en gran medida en el mundo de la programación, ya que es un entorno totalmente consolidado debido a que su lanzamiento se produjo en la década de los noventa. Además con el avance de los años han aparecido numerosas versiones por lo que es una herramienta que siempre está totalmente actualizada.

Entre esa multitud de versiones de Visual Studio existen también diferentes versiones que operan en diversos idiomas, este hecho facilita también la programación en gran medida a programadores inexpertos.

También **Visual Studio** es utilizado tanto en el mundo laboral y profesional como en la vida académica, ya que es muy común que se utilice para introducir a alumnos universitarios al mundo de la programación. Esto se debe tanto a su fiabilidad como a su versatilidad para programar en diferentes lenguajes [34], además aporta facilidad y claridad a la hora de programar.

Para realizar este TFG se han utilizado dos versiones de Visual Studio, en un primer momento se utilizó Visual Studio 2012 para empezar a programar, pero finalmente la que se utilizó para desarrollar todo el trabajo es Visual Studio 2010. Ambas versiones se obtuvieron de manera gratuita a través de la Universidad Carlos III, gracias a la colaboración que existe entre la universidad y Microsoft mediante el programa Microsoft Academic Alliance. Según este acuerdo Microsoft pone a disposición de estudiantes y profesores un gran número de programas y entornos de desarrollo de manera gratuita [35].

Capítulo 5. Ejecución en C

5.1. Lenguaje C

Durante la primera parte de este TFG se realizan una serie de implementaciones del algoritmo de cifrado AES en **lenguaje C**. Por este motivo a continuación se describen brevemente algunas de las características principales de este lenguaje de programación.

C es un lenguaje de programación desarrollado en 1972 por el científico Dennis Ritchie durante su estancia en los Laboratorios Bell. El lenguaje C de programación es una evolución del lenguaje B que existía anteriormente.

Apenas unos años más tarde comenzó a ser utilizado en gran medida llegando a remplazar a BASIC en algunas aplicaciones, su popularidad siguió aumentando hasta que fue estandarizado por las normas ANSI e ISO en 1989 y 1990 respectivamente [36].

Años más tarde el lenguaje C evoluciona hacia uno nuevo llamado C++, este nuevo lenguaje comparte muchas características con C, pero su principal diferencia es que C++ es está orientado a objetos [37].

A pesar de que C es uno de los primeros lenguajes de programación en aparecer, todavía hoy es ampliamente utilizado debido en parte a su sencillez a la hora de aprenderlo, por lo que es útil para aproximarse por primera vez al mundo de la programación.

Las principales características de C son [36] [37]:

- Es un lenguaje estructurado e imperativo y además admite el uso de estructuras de control.
- Es un lenguaje eficiente.
- Es una herramienta afable, potente y flexible a la hora de trabajar con él.
- Lo más importante es que es un lenguaje portable, es decir, puede ejecutarse prácticamente en cualquier ordenador con muy pocas modificaciones.
- Se pueden utilizar diferentes tipos de datos y funciones predefinidas a través de una serie de librerías.
- Es la base de C++.

Como ya se ha comentado anteriormente, las implementaciones de los códigos del TFG finalmente se desarrollaron en Visual Studio como herramienta de desarrollo, pero en un primer momento se comenzaron a realizar a través de otra plataforma más desconocida denominada **Qt Creator**.

Al igual que Visual Studio, Qt Creator es también un entorno de desarrollo integrado. Este entorno fue desarrollado por una compañía noruega vinculada a Nokia. Esta compañía es conocida como Qt Development Frameworks pero anteriormente era conocida como Trolltech [38].

Qt creator es un entorno similar a Visual Studio, pero a su vez es también un entorno más básico, simple y claro. Del mismo modo su interfaz ayuda y facilita en gran medida la edición del código y la corrección de errores.

Finalmente se decidió realizar la ejecución de las implementaciones del código mediante la plataforma **Visual Studio**, porque es un entorno más popular. Además cuenta con mayor prestigio y tradición en el mundo de la programación.

5.2. Código en C

En este apartado se describen las partes más importantes que componen el código del algoritmo de cifrado AES en el lenguaje C, así pues para su ejecución se utiliza solamente la CPU.

En primer lugar, se debe comentar que como el estudio se realiza con diferentes tamaños de texto plano no se analizan cada uno de los códigos ya que se trata de un proceso excesivamente repetitivo. Además en gran medida esas diferencias se limitan a pequeñas variaciones en los bucles. Por este motivo se analiza en profundidad un caso estándar, de esta manera se utiliza como texto plano los cuatro vectores de TEST del NIST.

Así pues, a continuación se analiza el código en C tanto para el proceso de cifrado como para el de descifrado. De todos modos se analizan las partes más importantes fragmentando el código, así que el código completo para el proceso de cifrado se puede consultar en el ANEXO A y el de descifrado en el ANEXO B.

En la siguiente figura se puede ver como se distribuye el código en C, es decir, la posición en la que se debe colocar el fragmento de código en función de su naturaleza.

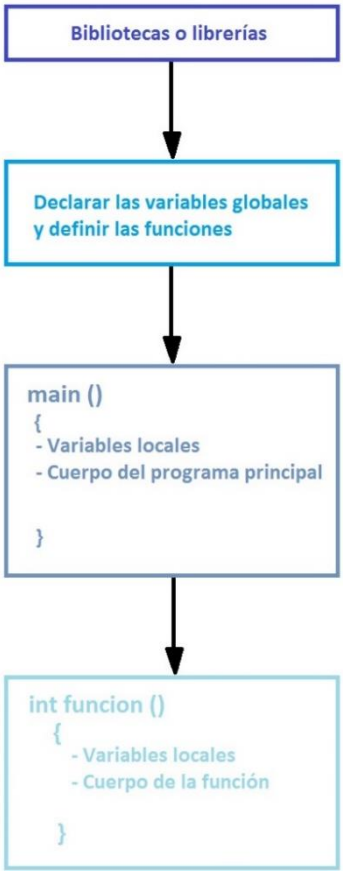

Figura 5.1 Esquema de un programa genérico en C

5.2.1. Bibliotecas o librerías

Al comienzo del código siempre deben aparecer las **librerías** que son necesarias para ejecutar el código. Las bibliotecas o librerías en programación son los ficheros que contienen las clases y funciones que se usan en la ejecución de los programas. En el caso de los programas creados en este TFG tanto para cifrar como para descifrar, se invocan solamente dos librerías:

```
#include <stdio.h>
#include <time.h>
```

La librería stdio.h es la librería estándar del lenguaje C, contiene las sentencias y definiciones básicas que cualquier programa puede poseer, por ejemplo contiene las funciones printf y scanf.

El cálculo del tiempo de ejecución del programa se realiza mediante la biblioteca time.h, este procedimiento se detalla más adelante.

Además también define el tamaño del texto plano en este caso son 64 componentes, lo que equivale a 512 bits. Concretamente se define de la siguiente manera:

```
#define N 64
```

5.2.2. Variables globales

Tras definir las librerías se declaran las **variables globales** que son las que se van a usar tanto en el main como en la funciones. En C también existen otro tipo de variables llamadas locales, estas variables solo son accesibles en el cuerpo de la función en la que son declaradas. En este caso se declaran diferentes variables como globales, son las siguientes:

```
int m;
unsigned int vector_claves [176];
unsigned int vector_estado[N]={
0x6b,0x2e,0xe9,0x73,0xc1,0x40,0x3d,0x93,0xbe,0x9f,0x7e,0x17,0xe2,0x96,0x11,0x2a,  //Vector 1
0xae,0x1e,0x9e,0x45,0x2d,0x03,0xb7,0xaf,0x8a,0xac,0x6f,0x8e,0x57,0x9c,0xac,0x51,  //Vector 2
0x30,0xa3,0xe5,0x1a,0xc8,0x5c,0xfb,0x0a,0x1c,0xe4,0xc1,0x52,0x46,0x11,0x19,0xef,  //Vector 3
0xf6,0xdf,0xad,0xe6,0x9f,0x4f,0x2b,0x6c,0x24,0x9b,0x41,0x37,0x45,0x17,0x7b,0x10}; //Vector 4
```

La variable m es de tipo entero e indica el número de vuelta por el que se va en el proceso de cifrado o descifrado. La variable denominada vector_claves es un array de 176 componentes, en él se almacenan todas las claves necesarias en el proceso, concretamente son once claves, los datos del array son de tipo entero positivo. Finalmente, la variable vector_estado es un array donde se almacena el texto plano, en este caso está formado por los cuatro vectores de TEST del NIST para el modo de codificar (es el que se muestra antes), para descodificar es equivalente pero con los valores descodificados, en ambos casos los datos son de tipo entero positivo.

Además en el modo codificar también se declara como variable global la matriz de sustitución de bytes, ya que se utiliza en el main para calcular todas las claves y en la función SubBytes. Pero para descodificar no se declara como variable global, porque solo se utiliza en el main.

En este campo también se deben declarar las funciones que más tarde se programan y utilizan. Las funciones se declaran en el campo de las variables globales, se llaman en el main, pero finalmente se programan fuera del main. De esta manera se añade el siguiente código para cuando se codifica:

```
void addRoundKey();
void subBytes();
void shiftRows();
void mixColumns();
```

De un modo análogo se declaran las siguientes funciones para el proceso inverso:

```
void addRoundKey();
void inv_shiftRows();
```

```
void inv_subBytes();
void inv_mixColumns();
```

5.2.3. Main

En primer lugar se debe explicar en qué consiste el main. El **main** es la función principal del programa, así pues en él se desarrolla el cuerpo del programa principal.

Al comienzo del main se declaran las variables locales, estas variables son aquellas que únicamente se pueden utilizan dentro de la función en la cual se declaran. Aunque en el main se pueden utilizar tanto estas funciones locales como las funciones globales declaradas anteriormente.

De este modo en el caso del AES se definen como variables locales las que son utilizadas en el cálculo de las subclaves (esta parte es común tanto para el proceso directo como para el inverso). Así se tiene lo siguiente:

```
int i,j,n,l, temporal;
	unsigned int clave[4][4]={
		{0x2b,0x28,0xab,0x09},
		{0x7e,0xae,0xf7,0xcf},
		{0x15,0xd2,0x15,0x4f},
		{0x16,0xa6,0x88,0x3c}
		 };
	unsigned int matriz_claves[4][44];
	 unsigned int sub_claves[4][8];
int Rcon [4][11]={
		 {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36},
		 {0X00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
		 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
		 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
		};
```

Las variables i, j, n, l y temporal son de tipo entero y se utilizan como variables auxiliares o contadores en los bucles. La matriz denominada clave contiene la clave de 128 bits elegida para cifrar o descifrar, en este caso es la clave de los vectores de TEST del NIST, es de tipo entero positivo. En el caso de matriz_claves se van almacenando las once claves a medida que se van calculando, al igual que en el caso anterior los datos son de tipo entero positivo. La matriz sub_claves se utiliza para almacenar y hacer las modificaciones necesarias para calcular cada clave. Finalmente se define Rcon que es la matriz de las potencias de dos. Estas dos últimas variables al igual que en los casos anteriores, son todas de tipo entero positivo.

En el caso del AES inverso también se definiría la matriz de sustitución de bytes (S-BOX), ya que previamente no se definió como global.

Después se procede a calcular las claves de ronda con las especificaciones correspondientes. Este procedimiento se puede consultar en los ANEXOS A y B.

A continuación se desarrolla el cifrado o descifrado propiamente dicho. Ambos procesos son muy similares, la diferencia es que para descodificar se utilizan las funciones inversas.

5.2.3.1. Cifrado con AES

Así pues en para **cifrar** mediante **AES** es necesario el uso de cuatro funciones, una por cada operación que se necesita para realizar el cifrado. También es necesario un bucle para realizar las distintas rondas. De esta manera se tiene el siguiente extracto de código para el proceso de cifrado:

```
m=0;
addRoundKey ();
for (l=0;l<=8;l++)
{
subBytes ();
shiftRows();
mixColumns();
m=m+1;
addRoundKey();
}
subBytes();
shiftRows();
m=m+1;
addRoundKey();
```

La variable m se utiliza para avanzar en el vector_claves y utilizar la clave correspondiente en cada ronda, además la variable l indica el número de ronda

De esta manera y tras mostrar por pantalla los resultados obtenidos, el main se termina con la instrucción:

```
return 0;
```

5.2.3.2. Descifrado con AES

De un modo totalmente análogo se procede para el proceso **inverso**, así pues se tiene el siguiente código:

```
m=10;
addRoundKey();
for(l=0;l<=8;l++)
{
inv_shiftRows();
inv_subBytes();
m=m-1;
addRoundKey();
inv_mixColumns();
}
inv_shiftRows();
inv_subBytes();
m=m-1;
addRoundKey();
```

Se puede observar que para realizar el descifrado son necesarias cuatro funciones, como ya se ha dicho estas son las funciones inversas. Además al igual que en el caso anterior también se utiliza la variable m para seleccionar la correspondiente clave del array vector_claves. Finalmente l se utiliza en el bucle para avanzar por las distintas rondas.

Al igual que en el proceso de cifrado, el main se termina mediante la siguiente sentencia:

```
return 0;
```

5.2.4. Funciones

Para la ejecución tanto del proceso de cifrado como descifrado se ha realizado mediante funciones creadas por el programador. Ya que el uso de funciones en programación permite dividir un programa de grandes dimensiones en pequeños fragmentos, esto aporta mayor claridad en el programa. Así pues, en el caso de este TFG se han creado siete funciones

en el lenguaje C para poder terminar el proceso de cifrado, todas estas funciones se declaran como void, esto quiere decir que no devuelven ningún valor, además al ser de tipo void no es obligatorio añadir la sentencia `return`. Las funciones se pueden declaran como void, porque las variables sobre las que actúa son globales, aunque también utilizan variables locales.

En los siguientes apartados se explican las funciones inversas y directas.

5.2.4.1. Funciones directas

A continuación se muestran las cuatro funciones que se utilizan en el proceso de cifrado, es decir la función **void addRoundKey (), void subBytes (), void ShiftRows ()** y por último la función **void mixColumns ().**

```
void addRoundKey ()
{ int i;
//Calculo la operación XOR entre el estado y la clave correspondiente
	for (i=0; i<16; i++)
	{
		vector_estado[i]=vector_estado[i]^vector_claves[(16*m)+i];
		vector_estado[i+16]=vector_estado[i+16]^vector_claves[(16*m)+i];
		vector_estado[i+32]=vector_estado[i+32]^vector_claves[(16*m)+i];
		vector_estado[i+48]=vector_estado[i+48]^vector_claves[(16*m)+i];
	}
}
```

La función **addRoundKey** () realiza la operación OR-Exclusiva entre el estado y la clave de ronda que corresponda, en esta función los cambios se ejecutan sobre la variable global vector_estado. Además esta función utiliza la variable entera i como un contador para avanzar a lo largo del array.

Esta función se utiliza tanto en el proceso directo como en el inverso, es decir no es necesario programar la función inversa de addRoundKey, porque la inversa del operador ⊕ (X-OR) es él mismo.

```
void subBytes()
{
	int i;
// Sustituyo cada elemento del estado por su equivalente en la matriz de sustitución de bytes
	for (i=0;i<N;i++)
	{
		vector_estado[i]=sbox[vector_estado[i]];
	}
}
```

En el caso de la función **subBytes** () se realiza una operación básica. El funcionamiento de esta función consiste en sustituir cada byte que conforma el estado por su equivalente en la matriz de sustitución de bytes (S-BOX).

Al igual que en la anterior función se utiliza la variable local i como contador, esta variable es declarada como entera. Se puede declarar de nuevo una variable local con el mismo nombre, porque las variables locales que se declaran dentro de una función no son reconocidas por otras funciones o por el main.

Además también se usa en la función subBytes el array sbox que en su momento se declaró como variable global, ya que previamente se usaba en el cálculo de todas las claves de ronda.

```
void shiftRows()
{
```

```
	int i;
	unsigned int estado_auxiliar [N];
	//Realizo las rotaciones circulares en las filas
	for(i=0;i<4;i++)
{
	//Primera fila
	estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
	estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
	estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
	estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
	//Segunda fila
	estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+5];
	estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+6];
	estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+7];
	estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+4];
	//Tercera fila
	estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
	estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
	estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
	estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
	//Cuarta fila
	estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+15];
	estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+12];
	estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+13];
	estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+14];
}
	for (i=0;i<N;i++)
	{
		vector_estado [i]=estado_auxiliar[i];
	}
}
```

En la función **shiftRows** () se realizan rotaciones circulares hacia la izquierda, cada fila rota un número diferente de posiciones. En una primera versión de este TFG esta función se implementó de otra manera, es decir, la fila cero no rotaba, pero la fila uno rotaba una posición, la fila dos se realizaba mediante un bucle que efectuaba dos veces esta misma operación y para la última fila se utilizaba un bucle para rotar tres posiciones. Así pues las rotaciones se realizaban mediante bucles, un bucle por cada fila de cada estado.

En la versión intermedia las rotaciones se realizan mediante un estado intermedio. Este estado intermedio se declara como un array de tipo positivo con 64 componentes, se denomina estado_auxiliar y permite realizar las rotaciones en cada estado en un mismo bucle, de esta manera se reduce considerablemente el número de bucles que se utilizan en esta función.

Tras utilizar este estado intermedio en la transformación, los resultados se escriben en el vector_estado original para poder seguir con el proceso.

```
void mixColumns ()
{
   unsigned char Tmep,Tme,time;
// Multiplico cada columna por una matriz concreta en el Campo de Galois
// xtime es un macro que devuelve el producto con modulo {1b} de {02} y el byte argumento
    #define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

	int i,j;
	for (i=0;i<4;i++)
	{
	   for (j=0;j<4;j++)
	  {
```

```
time=vector_estado[(16*i)+j];
Tmep = vector_estado[(16*i)+j] ^ vector_estado[(16*i+4)+j] ^
vector_estado[(16*i+8)+j] ^ vector_estado[(16*i+12)+j] ;
Tme = vector_estado[(16*i)+j] ^ vector_estado[(16*i+4)+j] ;
Tme = xtime(Tme);
vector_estado[(16*i)+j] ^= Tme ^ Tmep ;
Tme = vector_estado[(16*i+4)+j] ^ vector_estado[(16*i+8)+j] ;
Tme = xtime(Tme);
vector_estado[(16*i+4)+j] ^= Tme ^ Tmep ;
Tme = vector_estado[(16*i+8)+j] ^ vector_estado[(16*i+12)+j] ;
Tme = xtime(Tme);
vector_estado[(16*i+8)+j] ^= Tme ^ Tmep ;
Tme = vector_estado[(16*i+12)+j] ^ time ; Tme = xtime(Tme);
vector_estado[(16*i+12)+j] ^= Tme ^ Tmep ;
                                    }
       }
}
```

La función **mixColumns** () es la última función del proceso directo de cifrado. Esta función multiplica cada columna del estado por una determinada matriz en el Campo de Galois, con todas las premisas que estos campos tienen.

Se debe decir que para la realización de esta función se parte del código programado por Eduardo Bonilla para la realización de su PFC [39]. En ella se declaran las variables enteras i y j como temporales y las variables de tipo carácter Tmep, Tme y time como auxiliares. Además también se define el macro `xtime`, que realiza desplazamientos y multiplicaciones dentro de cada componente del estado para que siempre se tengan dos cifras por componente.

5.2.4.2. Funciones inversas

En este apartado se describen las funciones inversas que se utilizan para realizar el proceso inverso, es decir, para descodificar un mensaje cifrado. A continuación se muestran las funciones **void inv_shiftRows (), void inv_subBytes ()** y **void inv_mixColumns ().**

```
void inv_shiftRows()
{
int i;
  unsigned int estado_auxiliar[N];
//Realizo las rotaciones circulares en el sentido opuesto
  for(i=0;i<4;i++)
{
       //Primera fila
       estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
       estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
       estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
       estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
       //Segunda fila
       estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+7];
       estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+4];
       estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+5];
       estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+6];
       //Tercera fila
       estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
       estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
       estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
       estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
       //Cuarta fila
       estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+13];
       estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+14];
       estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+15];
       estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+12];
```

```
}
	for(i=0; i<N; i++)

	{
		vector_estado[i]=estado_auxiliar[i];
	}
}
```

La función **inv_shiftRows** realiza rotaciones circulares, pero en esta ocasión los desplazamientos se producen hacia la derecha. Al igual que para la función directa, primero se implementó utilizando un bucle diferente por cada fila.

Finalmente esta función se programa utilizando un estado intermedio (estado_auxiliar) de un modo totalmente análogo a como se usa para la función shiftRows. Así pues, solamente se declaran como variables locales las siguientes: i como contador y el array de 64 componentes estado_auxiliar.

```
void inv_subBytes()
{	int i;
//Sustituyo cada elemento de la matriz por su equivalente en la matriz de sustitución inversa
//Tabla inversa de sustitucion bytes (iS-box)
    int isbox[256] =     {
    //0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, //0
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, //1
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, //2
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, //3
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, //4
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, //5
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, //6
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, //7
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73, //8
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, //9
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, //A
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, //B
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f, //C
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, //D
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, //E
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};//F
	for(i=0;i<N;i++)
	{
		vector_estado[i]=isbox[vector_estado[i]];
	}
}
```

El funcionamiento de la función **inv_subBytes** consiste simplemente en sustituir cada componente del estado por su equivalente en la matriz de sustitución de bytes inversa.

Como se puede comprobar su comportamiento es similar al de la función subBytes, la única diferencia es que en este caso se utiliza la matriz de sustitución inversa. Además esta matriz se declara como local dentro de la función ya que no se utiliza en ningún otro lugar del programa.

```
void inv_mixColumns()
{
	int i,j,auxiliar;
	#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))
	//Multiplico cada columna por su inversa en el Campo de Galois
	 // Multiply es un macro que multiplica numeros en el gampo GF(2^8)
	 int num_a,num_b,num_c,num_d;
	 #define Multiply(x,y) (((y & 1) * x) ^ ((y>>1 & 1) * xtime(x)) ^ ((y>>2 &
1) * xtime(xtime(x))) ^ ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^ ((y>>4 & 1) *
xtime(xtime(xtime(xtime(x))))))
		for(i=0;i<4;i++)
		{
			for(j=0;j<4;j++)
			{
```

```
num_a = vector_estado[(16*i)+j]; num_b = vector_estado[(16*i+4)+j];
num_c = vector_estado[(16*i+8)+j]; num_d = vector_estado[(16*i+12)+j];
vector_estado[(16*i)+j] = Multiply(num_a, 0x0e) ^ Multiply(num_b, 0x0b) ^
Multiply(num_c, 0x0d) ^ Multiply(num_d, 0x09);
vector_estado[(16*i+4)+j] = Multiply(num_a, 0x09) ^ Multiply(num_b, 0x0e) ^
Multiply(num_c, 0x0b) ^ Multiply(num_d, 0x0d);
vector_estado[(16*i+8)+j] = Multiply(num_a, 0x0d) ^ Multiply(num_b, 0x09) ^
Multiply(num_c, 0x0e) ^ Multiply(num_d, 0x0b);
vector_estado[(16*i+12)+j] = Multiply(num_a, 0x0b) ^ Multiply(num_b, 0x0d) ^
Multiply(num_c, 0x09) ^ Multiply(num_d, 0x0e);
                    }
                }
                for(i=0;i<N;i++)
                {
auxiliar=vector_estado[i]/0x100;vector_estado[i]=vector_estado[i]-auxiliar*0x100;
                }
}
```

Al igual que en el caso de la función mixColumns, para implementar la función **inv_mixColumns** se parte del código de Eduardo Bonilla [39].

En esta función se multiplica cada columna por una matriz en el Campo de Galois, para ello se utilizar las variables locales y enteras i, j y auxiliar, i y j se utilizan en los bucles y auxiliar se utiliza para obtener dos dígitos en casa byte y seguir operando en hexadecimal. Además también se define el mismo macro `xtime` que en la función mixColumns, pero también se define el macro `Multiply` para realizar las multiplicaciones.

Como se puede comprobar, la programación de las funciones inversas es muy similar a las funciones del proceso directo, ya que solamente hay que implementar una serie de modificaciones en cada función.

5.2.5. Cálculo de tiempo

Con el fin de comparar el resultado de la ejecución del programa tanto en C como en CUDA, se debe medir el tiempo que tarda en ejecutarse el código. Para ello se usa la librería `<time.h>`. Hay que declarar dos variables tipo `clock_t` de esta manera se tiene una variable para iniciar el evento y otra para finalizarlo, además también es necesario otra función de tipo `double` para registrar la resta de estos eventos [40]. Este concepto se puede ver de un modo genérico a continuación:

```
//Defino las variables necesarias
clock_t begin, end;

double time_spent;

//Inicio el contador de tiempo
begin = clock();

...
Realizo la ejecución del programa
...

//Finalizo el contador de tiempo
end = clock();
//Calculo el tiempo de ejecución
time_spent = (double) (end - begin) / CLOCKS_PER_SEC;
```

Además se utiliza la sentencia `CLOCKS_PER_SEC` que se refiere al número de pulsos de reloj por segundo.

En el caso del algoritmo AES, el contador se inicia justo al comenzar el proceso de cifrado, es decir, justo después de finalizar el cálculo de las subclaves. Además el otro contador se coloca nada más terminar el proceso de cifrado. De esta manera se temporiza lo que tarda el sistema en cifrar o descifrar, sin tener en cuenta el cálculo de las claves de ronda. Solamente se temporiza ese fragmento de código, porque esa es la parte de código que se paraleliza mediante CUDA.

Finalmente se debe destacar, que esta función calcula el tiempo en segundos, por lo que en algunas ocasiones, sobre todo para valores pequeños los datos obtenidos no son muy exactos, sino más bien se trata de valores aproximados.

5.2.6. Cifrado y descifrado con otros tamaños de texto plano

En el presente TFG, se realizan estudios y pruebas con textos planos de diferentes tamaños, ya que uno de los objetivos es estudiar la evolución de CUDA con textos de distintas dimensiones.

Por este motivo se deben hacer modificaciones en las líneas de código, estas modificaciones se explican someramente sobre los códigos que aparecen en los ANEXOS A y B, ya que en dichos anexos se explica detenidamente el proceso de cifrado y descifrado para un texto plano de 64 componentes.

Simplemente hay que cambiar el valor que define N en la sentencia `#define N` al comienzo del código, del mismo modo al definir el array llamado `unsigned int vector_estado[N]` se deben declarar tantas componentes como el valor numérico de N. Por lo demás el main del código no sufre ninguna modificación.

Aunque la mayoría de las funciones si sufren pequeñas modificaciones. Estas pequeñas correcciones se limitan a cambiar los valores en los bucles, para que las funciones recorran completamente la variable vector_estado y poder finalizar así correctamente el proceso de cifrado o descifrado.

Capítulo 6. Ejecución en CUDA

6.1. Introducción

A lo largo del presente capítulo, se hace una introducción a la programación en **CUDA**, así como al mundo de las **GPU** y a su uso en programación debido a la gran potencia de cálculo que ofrecen.

Además también se comenta y explica el funcionamiento de las principales instrucciones de **CUDA**. Finalmente se explica el código implementado destacando sus partes más importante facilitando así su comprensión.

6.2. GPU

El elemento indispensable para la ejecución de CUDA son las GPUS. Las tarjetas gráficas también se conocen como **GPU** debido a las siglas de su denominación inglesa Graphics Processing Unit.

Las actuales y potentes GPUS tienen su origen en los años 60 con la aparición de las primeras tarjetas gráficas que comenzaban a surgir con los primeros monitores de la época. Pero es con la generalización del uso del PC cuando verdaderamente comienzan a tomar la importancia que merecen. Así pues, Poco a poco y al igual que en el resto de tecnología, las tarjetas gráficas comenzaron a avanzar en torno a la visualización de dos dimensiones y a un mayor espectro cromático. De este modo, también comienza una cierta revolución. Hasta llegar al momento actual donde las GPUS tienen una gran potencia de cálculo y están optimizadas para el cálculo con valores en coma flotante, este tipo de datos es utilizado en los gráficos en 3D [41].

Muchas de esas aplicaciones gráficas presentan un gran nivel de paralelismo, por lo que es una buena solución utilizar esa gran capacidad de cálculo de las GPU, para ejecutar cálculos al mismo tiempo.

Las **GPUS** actuales presentan mayor cantidad de vértices y píxeles. Así pues, aunque su frecuencia de reloj sea menor a la de las CPU, la potencia de cálculo en paralelo es mucho mayor debido a la estructura de sus componentes. Además, también se debe destacar que la CPU presenta una arquitectura Von Neumann, ya que almacena los datos y las instrucciones en el mismo dispositivo, mientras que la GPU utiliza el modelo circulante para facilitar la ejecución de proceso en paralelo [41].

En el capítulo anterior se utiliza la **CPU** para implementar los códigos en C, mientras que en este capítulo se utiliza la potencia de la **GPU**. La principal diferencia entre la CPU y la GPU es que la CPU posee unos cuantos núcleos optimizados para la ejecución secuencial, mientras que la GPU tiene miles de núcleos, más eficientes y pequeños que están diseñados especialmente para procesar tareas de manera simultánea. Así pues, las GPUS presentan miles de núcleos, gracias a los cuales se procesan las cargas de trabajo de un modo eficiente, porque las procesa en paralelo [42].

Esta gran diferencia se puede ver fácilmente a través de la siguiente figura anexa. En ella se puede observar la gran variación que existe entre el número de núcleos que poseen la CPU y la GPU.

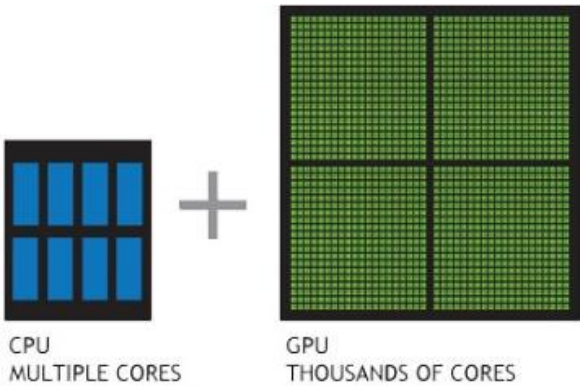

Figura 6.1 Diferencia entre CPU y GPU [42]

Debido a esta gran capacidad y potencia de cálculo las GPUS se utilizan actualmente para muchas más aplicaciones que poder ver un video o jugar a un videojuego con una mayor definición de imagen y gráficos, ya que también son utilizadas en multitud de aplicaciones por investigadores para acelerar la ejecución de cálculos. Este caso es el que nos ocupa en la realización del presente trabajo.

Como ya se ha comentado anteriormente, para la ejecución de este proyecto se utiliza una **Nvidia GeForce GTX 550 Ti** como GPU.

Figura 6.2 Tarjeta Nvidia GeForce GTX 550 Ti [43]

6.2.1 GPGPU

Relacionado con la aparición de la GPU, también aparece otra nueva denominación que es la técnica **GPGPU**. Estas siglas corresponden a General Purpose Computing on Graphics Processing Units, también se puede denominar como GPU Computing [41]. El propósito de esta nueva disciplina es utilizar la potencia de cálculo de las tarjetas gráficas para cualquier otro ámbito, que no tiene por qué estar relacionado con los gráficos.

En primer momento, llevar a la práctica esta teoría era muy difícil, ya que no existían programas y herramientas de desarrollo que estuviesen preparadas para facilitar la programación paralela a través de la GPU. Aunque actualmente este problema ya no existe, porque existen varias herramientas que permiten tomar el control de la GPU para programar aplicaciones utilizando su potencia.

De este modo, actualmente la aplicación de **GPGPU** está muy extendida en el mundo científico, pero también esta teoría se puede utilizar en aplicaciones de uso cotidiano.

6.2.2. Herramientas de GPGPU

Actualmente existen multitud de empresas dedicadas a la fabricación de GPU. Entre esas empresas destacan **AMD** (desde que adquirió la antigua ATI) y **Nvidia** por ser las más populares e importantes. En los últimos años, ambas empresas han desarrollado plataformas para desarrollar la teoría del **GPGPU**. Estas herramientas permiten utilizar la programación paralela. De este modo se pueden diferenciar de la competencia a la vez que impulsan una nueva línea comercial ya que pueden ampliar así su espectro de clientes, aumentando fuertemente su número de ventas.

Con esta filosofía AMD desarrolla **Stream SDK**, mientras que Nvidia trabaja con **CUDA**, esta última plataforma es la que se va a utilizar para desarrollar este TFG. Se utiliza CUDA porque es una herramienta más popular a la hora de paralelizar algoritmos y existen muchos más investigaciones que utilizan esta plataforma como elemento de cálculo en el ámbito científico, así pues es una herramienta que tiene un uso más extendido. Aunque se profundiza más sobre CUDA en capítulos posteriores, ya que es la herramienta utilizada.

A su vez existen otro tipo de tecnologías que no se limitan a trabajar en exclusividad con un único fabricante de GPU. Este es el caso de OpenCL, que aparece con la idea de convertirse en un estándar, además es de código abierto y gratuito. OpenCL permite programar aplicaciones de ejecución paralela con el uso de la GPU y de un lenguaje concreto, la principal ventaja de este sistema es que no se restringe a un único fabricante de tarjetas [41]. Por este motivo OpenCL ya es aceptada por las compañías más importantes del sector, como es el caso de Nvidia, AMD, Intel, Apple,… Aunque también se debe decir que aunque esta plataforma sea muy potente y este llamada a convertirse en un estándar en el futuro, a día de hoy todavía está en proceso de desarrollo.

6.3. CUDA

De esta manera, la plataforma utilizada para desarrollar esta parte del TFG es CUDA. Las siglas de **CUDA** corresponden a Compute Unified Device Arquitecture. Como ya se ha comentado CUDA es una arquitectura desarrollada por Nvidia para realizar programación paralela mediante el uso de sus GPUS. La aplicación de esta tecnología debe proporcionar un incremento extraordinario del rendimiento del sistema [44].

CUDA ha supuesto una auténtica revolución en el mundo de la programación, por este motivo se han multiplicado tanto las investigaciones científicas como los cursos en universidades en los últimos años. Este aumento se puede ver en los siguientes datos:

Tabla 6.1 Evolución de CUDA [45]

	Año 2008	Año 2014
GPUs con CUDA	100.000.000	500.000.000
Descargas de CUDA	150.000	2.100.000
Superordenadores	1	52
Cursos universitarios	60	780
Trabajos académicos	4.000	40.000

Con estos sencillos ejemplos se puede comprobar la fuerte evolución que ha sufrido CUDA en los últimos años. Esto es debido a que CUDA es una arquitectura muy potente y con múltiples posibilidades, por este motivo esta herramienta está todavía en constante expansión y cada día sus descargas aumentan en número.

De este modo, la tendencia actual es abandonar el procesamiento central en la CPU y dirigirse hacia un coprocesamiento entre la CPU y la GPU. Para facilitar el uso de esta nueva tendencia, Nvidia incluye de serie esta tecnología en todas las GPUs GeFoce, ION Quadro y Tesla GPUs, lo que facilita y acerca está tecnología en gran medida a los programadores [44].

Actualmente existen multitud de aplicaciones que ya se aceleran mediante CUDA, este es el caso de muchas aplicaciones de video, pero también de aplicaciones en el ámbito científico y financiero entre otros.

Se puede comprobar fácilmente observando las figuras 6.3 y 6.4 la potencia que posee CUDA En ellas se puede ver la comparativa del poder de cómputo y el ancho de banda de memoria, ambos teóricos, entre la GPU y la CPU:

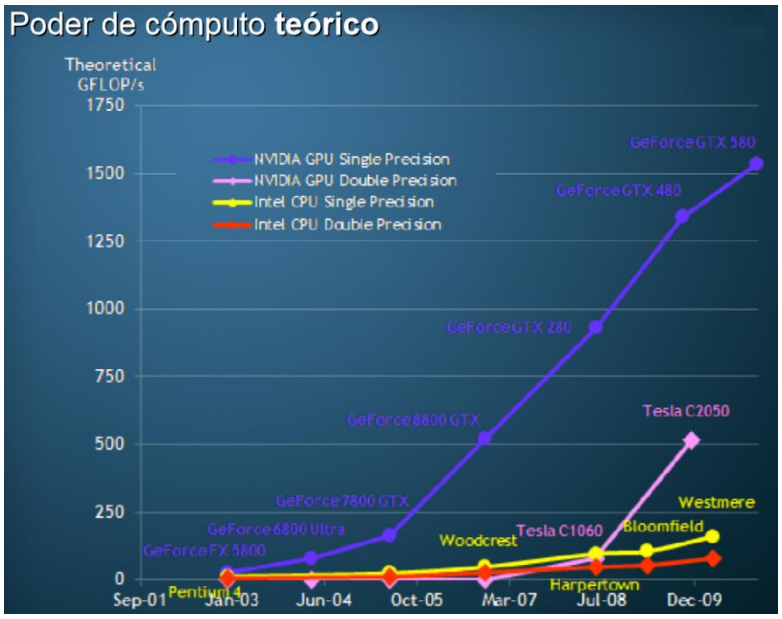

Figura 6.3 Evolución del poder de cómputo teórico en las CPU y las GPU [46]

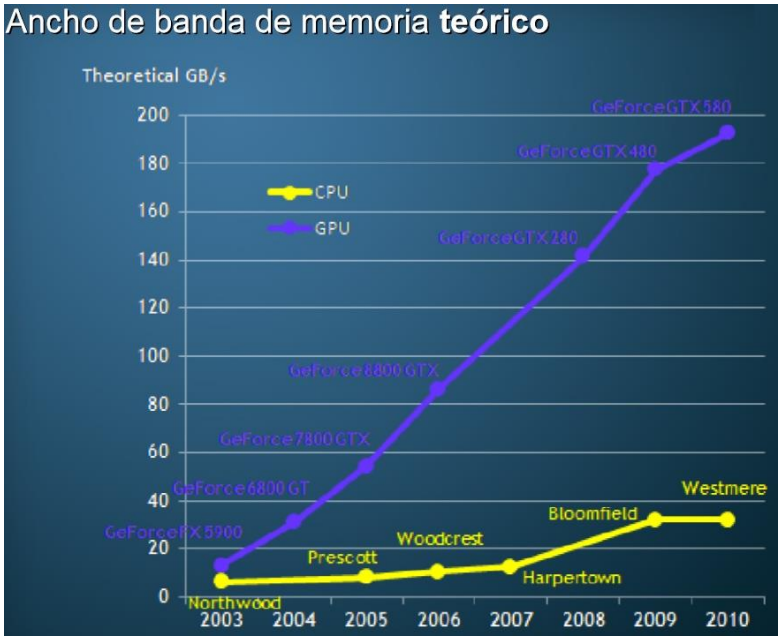

Figura 6.4 Evolución del ancho de banda teórico en las CPU y las GPU [46]

A la vista de los resultados de las gráficas anteriores, se comprueba como el potencial de las GPU es mucho mayor.

Para programar en CUDA se utilizan una serie de extensiones de C y C++. Además el programador puede expresar el paralelismo mediante **Fortran, C o C++** como lenguajes de alto nivel, o incluso mediante estándares abiertos como en el caso de **OpenACC** [44].

En el caso de este TFG se programa en CUDA a partir del lenguaje de programación C, es decir, como una extensión. La principal característica de CUDA es que permite definir funciones en C, que se llaman **kernels** y son ejecutadas N veces en paralelo para N hilos diferentes [47].

Así pues, combina el uso de la CPU y la GPU. Porque en los programas en CUDA aparece una parte que está escrita en un lenguaje de programación tradicional, en el caso de este TFG en lenguaje C, esta parte se ejecuta a través de la CPU y en ella se invoca al Kernel. Además aparece otra parte que es en la que se declara el Kernel, esta parte se ejecuta de forma paralela mediante la GPU, para ello se programa a través de CUDA, que en el fondo es lenguaje C con ciertas modificaciones para regular la paralelización. Del mismo modo, se puede programar funciones que se ejecutan en la GPU o en la CPU, en función del diseño del programa.

6.3.1. Definiciones básicas en CUDA

En el presente apartado se definen brevemente una serie de conceptos claves para comprender mejor CUDA:

- **Threads**: también se conocen como hilos, son las unidades básicas de ejecución en CUDA. Cada hilo ejecuta un trabajo sobre cada uno de los elementos que componen el objeto inicial, por ejemplo en una matriz cada hilo corresponde a una componente [47].
- **Warps**: los threads se agrupan en warps. En un warps los hilos se ejecutan en paralelo, se empiezan a ejecutar en el mismo instante y siguen las mismas líneas de código. Aunque también pueden seguir caminos diferentes, pero en estos casos para terminar la ejecución del código se debe esperar a que terminen todos los threads [41].
- **Bloques**: también son un conjunto de hilos, pero a un nivel superior, a su vez componen los grid. Los bloques se tratan de lanzar de modo paralelo, aunque el tamaño del bloque puede ser mayor que en número de hilos que se puede utilizar paralelamente, pero la GPU los puede ejecutar por warps [47].
- **Grid**: cada grid se compone por bloques. Es el conjunto de bloques que se ejecuta en un kernel. Así pues los kernels se lanzan en grid [48].
- **Host**: es como se conoce a la CPU (Central Processing Unit).
- **Device**: hace referencia a la GPU (Graphics Processing Unit).
- **Kernel**: es una función que se invoca en el host, pero se ejecuta en el device. El kernel no se ejecuta en modo secuencial, sino que se ejecuta en N hilos distintos.

En capítulos posteriores se ejemplifica el uso de estos términos mediante extractos de código, así pues se muestra como se declaran algunos términos y en qué contexto se usan.

Para obtener un buen rendimiento en los kernel y programas en CUDA, se debe perseguir los siguientes ítems [47]:

- Hacer un uso eficiente de las memorias.
- Maximizar el paralelismo mediante los recursos disponibles.

- Explotar la localidad de los datos.

6.3.2. Jerarquía de hilos

CUDA tiene la capacidad de administrar los **hilos** en los que el programa de ejecuta, para ello cada hilo se identifica a través de un índice. Los índices pueden ser unidimensionales, bidimensionales o tridimensionales, de esta manera se forman bloques de una, dos y tres dimensiones, según las necesidades concretas del problema a resolver. Así pues, los hilos se identifican como `threadIdx.x, threadIdx.y,` y `threadIdx.z,` cada índice del hilo muestra una dimensión distinta.

Los hilos dentro de un mismo bloque cooperar entre ellos, de esta manera comparten datos a través de la memoria compartida, además se sincronizan mediante la ejecución coordenada de accesos de memoria [47].

Los bloques de hilos se ejecutan independientemente, se pueden ejecutar en paralelo o serie y en cualquier orden. Lo que permite que los bloques de hilos se organicen de cualquier forma en los núcleos. Además el número de hilos por bloque está restringido por la memoria de cada núcleo del procesador [47].

6.3.3. Jerarquía de memoria

En CUDA todos los hilos tienen la capacidad de acceder a múltiples espacios de memoria. De esta manera, cada hilo presenta una memoria local privada y cada bloque tiene una memoria compartida que es visible para todos los hilos de ese bloque. Además todos los hilos pueden acceder a la memoria global.

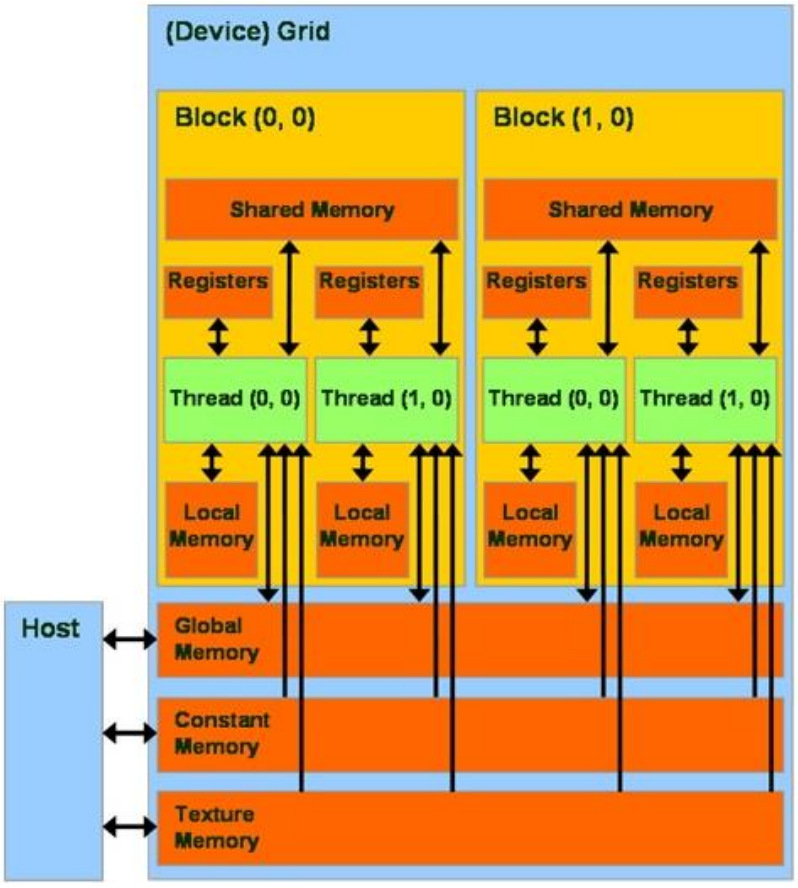

Figura 6.5 Tipos de memoria en CUDA [52]

En la anterior figura se puede ver los diferentes tipos de memoria, así como su ámbito de actuación. A continuación se explican someramente las características de las distintas memorias que aparecen en CUDA [46]:

- Cada thread tiene una **memoria privada** que es almacenada en registros. Los registros son el tipo de memoria más rápido
- La **memoria compartida** (shared memory) permite compartir los datos entre lo threads de un mismo bloque. Permite la sincronización de los hilos de un mismo bloque.
- La **memoria global** (global memory) es la memoria RAM de la GPU, se utiliza para cargar los datos. Es accesible para todos los threads. Puede ser el cuello de botella en los programas.
- La **memoria constante** (constant memory), es una memoria de solo lectura, se puede cargar desde el host o el device. Normalmente se utiliza para almacenar la información que es invariante a todos los hilos.
- La **texture memory** es también de solo lectura, se utiliza en procesamientos de video e imágenes. Es accesible para todos los hilos.

Las transferencias entre CPU y GPU son las más costosas de realizar, así pues estas transacciones se deben minimizar. El acceso de memoria es el cuello de botella, por lo que es más efectivo recalcular datos que almacenarlos en la memoria [46].

Además también hay que tener en cuenta que es preferible una transferencia grande entre la CPU y la GPU que efectuar muchas transferencias pequeñas. Del mismo modo, es posible que algunos algoritmos no tengan una implementación óptima en la GPU y sea mejor implementarlos en la CPU. Finalmente también se debe tener en cuenta que si el número de operaciones por dato es bajo, el tiempo de ejecución será dominado por el traspaso de memoria entre la CPU y la GPU [46].

Así pues, cuando se programa en CUDA hay que conocer los diferentes tipos de memoria así como sus principales características. Además también se debe tener en cuenta las premisas que se acaban de comentar sobre los traspasos de memoria. En la práctica las memorias del host y el device son diferentes, por este motivo que los datos o información se deben traspasar de un dispositivo a otro.

6.3.4. Arquitectura SIMT

Para que CUDA pueda manejar cientos de hilos a lo largo de distintos programas, utiliza una arquitectura conocida como **SIMT**. Este sistema asigna a cada thread un núcleo del procesador y cada hilo se ejecuta de modo independiente con sus propias instrucciones y registros [47].

Así pues, La ejecución de CUDA se sustenta sobre la arquitectura **SIMT** (Single Instruction Multiple Threads), el funcionamiento de esta arquitectura es similar a la de la teoría SIMD, pero utiliza hilos en lugar de una organización vectorial. Además en **SIMT** con una única instrucción se pueden controlar diferentes elementos a procesar [47].

La principal diferencia es que **SIMT** detalla la ramificación y ejecución de un único hilo, mientras que SIMD expone el ancho del campo vectorial al software. Además SIMD permite escribir código paralelo a nivel de hilo al programador [47]. Así pues en el procesador vectorial no hay ninguna diferencia en las ejecuciones paralelas, en cambio con el SIMT cada hilo puede tomar un camino distinto e independiente [41].

6.4. Implementaciones de AES en CUDA

Existen varias investigaciones científicas que han implementado el algoritmo AES sobre plataforma **CUDA**, pero los resultados obtenidos han sido muy dispares. Porque en algunas ocasiones se han obtenido buenos resultados acelerando el algoritmo respecto al tiempo de computo en C, mientras que en otras no se ha conseguido que la ejecución en CUDA sea más rápida.

Principalmente existen dos factores que justifiquen esa disparidad en los resultados. Quizás el factor más limitante sea la capacidad que posee cada **GPU** utilizada, ya que en cada estudio se utiliza una distinta con unas características diferentes.

El otro principio es que algunas de esas investigaciones no implementan la versión **estándar** del AES reconocida por el NIST, es decir, no se utilizan los cuatro tipos de transformación (SubBytes, ShiftRows, MixColumns y AddRoundKey). Por ejemplo este otro tipo de implementación no utiliza la S-BOX, sino que los procesos de cifrado y descifrado se realizan mediante otra matriz llamada T-BOX y por tanto también se producen algunas variaciones en las etapas de cada ronda. Pero en el caso de este TFG la implementación que se realiza del algoritmo AES es la estándar reconocida el NIST, ya que es la oficial y la más importante.

Entre esa disparidad de resultados destaca la investigación de Keisure Iwai y Takakazu Kurokawa [49] que obtuvo una velocidad 10 veces mayor con una Nvidia GeForce GTX 285 que con un corei7. Otra investigación es la de Qinjian Li y Chengwen Zhong [50] que consiguió una aceleración 50 veces mayor con Nvidia Tesla C2050. Ambas investigaciones utilizan la implementación del AES mediante la T-BOX.

Además existe otra investigación que compara la implementación del AES con la S-BOX y la T-BOX [51], en ella se observa como el rendimiento es mejor si se aplica la S-BOX en lugar de la matriz T-box, pero en este caso AES no se implementa sobre GPU.

Tras analizar estos trabajos previos, se decide implementar sobre la GPU la versión estándar del AES reconocida por el NIST en el FIPS 197 [19]. Se decide implementar esta opción porque el objetivo principal de este trabajo es conocer el comportamiento del algoritmo AES en su versión estándar sobre GPU, paralelizándolo así mediante CUDA. Así pues, este trabajo se puede ver cómo un estudio de viabilidad de la implementación de AES sobre GPU.

Además, se decide realizar la implementación sobre una GPU de bajo coste, que presenta unas características más dicretas que las de la tecnología Tesla. Las investigaciones comentadas anteriormente utilizan la arquitectura Tesla como GPU y esta tecnología presentan mejores prestaciones que las GeForce, pero también tienen un coste muy superior. De este modo se pretende observar la actuación del AES en una GPU asequible y que cualquier persona podría tener en su ordenador personal.

6.5. Código en CUDA

En el presente apartado se detallan las principales partes que componen el código del algoritmo **AES** tanto para cifrar como para descifrar en lenguaje **CUDA**. En el caso de la implementación mediante CUDA se utilizan la GPU y la CPU, por lo que una parte del código se programa en lenguaje C tradicional (la que se ejecuta en la CPU) y la otra en CUDA (la que se ejecuta mediante la CPU).

Al igual que en el caso del programa en C, se realizan diferentes ensayos con diferentes tamaños de texto plano, pero el proceso que se explica con mayor detalle es el de cifrar y

descifrar los cuatro vectores de TEST del NIST. Aunque para variar el tamaño del texto plano solamente se deben realizar pequeñas modificaciones en el código.

Así pues, en este apartado se analizan las partes más importantes del código de AES en CUDA para el proceso de cifrado y descifrado. Aunque el código completo se puede consultar en los ANEXOS C y D.

En la próxima figura se puede consultar la estructura de un código genérico en CUDA, en ella se ve por ejemplo donde se debe definir e invocar un kernel.

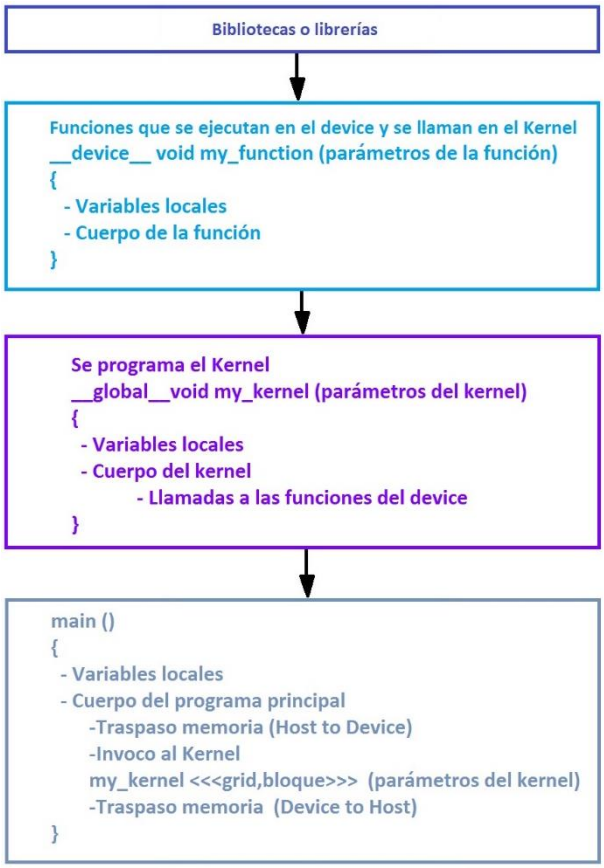

Figura 6.6 Esquema de un programa genérico en CUDA

En este apartado también se compara el código en CUDA con el de C, porque aunque las modificaciones que se realizan en este código puedan parecer pequeñas, son muy importantes y están colocadas en puntos estratégicos.

Además según se analiza el código se explica el funcionamiento de las instrucciones propias de CUDA que se utilizan.

6.5.1. Bibliotecas o librerías

Al igual que en el lenguaje en C, al comienzo del código en CUDA aparecen las **librerías**. Estas librerías son esenciales para poder ejecutar los códigos, ya que contiene ficheros y funciones necesarios par la ejecución de los programas. De esta manera aparecen las siguientes librerías:

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
```

El funcionamiento de estas librerías ya se ha comentado en su momento, <stdio.h> es la librería estándar y contiene las sentencias básicas para ejecutar un programa. La librería <time.h> se utiliza para calcular el tiempo de ejecución. Además se utiliza otra librería, <Windows.h>, para calcular el tiempo con otra precisión, su uso se datalla más adelante.
Finalmente también se define el número de componentes que forman el texto plano, en este caso 64 componentes o lo que es lo mismo, 512 bits.

```
#define N 64
```

CUDA también posee librerías propias, pero en este caso no se utilizan.

6.5.2. Main

El **main** es la función principal del programa. El main se programa en un lenguaje de programación tradicional, en el caso de este trabajo se programa en C. La característica fundamental de esta parte del código es que se ejecuta de modo secuencial y se procesa mediante la **CPU**.

Además aunque el main se ejecuta en la CPU, en el se reserva la memoria que se utiliza en la GPU, así como las llamadas al **kernel** y los traspasos de memoria entre dispositivos. Pero estoa conceptos se van detallando amedida que se avanza en el código.

Para concluir, se debe comentar que el proceso que se detalla corresponde al modo de cifrado, ya que el proceso de descifrado es totalmente análogo y se puede consultar fácilmente en el ANEXO D. Además en este análisis también se omiten algunas partes del código, por considerar irrelevante su relación con CUDA, así pues para conocer el código completo se remite a los ANEXOS C y D.

También se debe decir que en el proceso que se detalla solamente se usa la memoria global, pero más tarde se realizan otras implementaciones con otras características que se explican en el correspondiente momento.

En primer lugar, se definen las variables locales que se usan en la función:

```
// Variables que se utilizan como contadores
int i ,j ,n , temporal;
unsigned int vector_estado[N]={
0x6b,0x2e,0xe9,0x73,0xc1,0x40,0x3d,0x93,0xbe,0x9f,0x7e,0x17,0xe2,0x96,0x11,0x2a,0
xae,0x1e,0x9e,0x45,0x2d,0x03,0xb7,0xaf,0x8a,0xac,0x6f,0x8e,0x57,0x9c,0xac,0x51,0x
30,0xa3,0xe5,0x1a,0xc8,0x5c,0xfb,0x0a,0x1c,0xe4,0xc1,0x52,0x46,0x11,0x19,0xef,0xf
6,0xdf,0xad,0xe6,0x9f,0x4f,0x2b,0x6c,0x24,0x9b,0x41,0x37,0x45,0x17,0x7b,0x10};
//Clave para codificar los vectores de TEST
unsigned int clave[4][4]={
 {0x2b,0x28,0xab,0x09},
 {0x7e,0xae,0xf7,0xcf},
 {0x15,0xd2,0x15,0x4f},
 {0x16,0xa6,0x88,0x3c}
        };
unsigned int matriz_claves[4][44];
//En el vector de claves se almacenan todas las claves,para descodificar las ultimas
16 componentes son la clave de descifrado,es decir, se colocan en sentido inverso
//Una vez calculadas todas las claves las meto en un vector para que sea más
cómodo trabajar en la GPU
unsigned int vector_claves [176];
unsigned int sub_claves[4][8];
int Rcon [4][11]={
                {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36},
                {0X00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
```

```
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
               };
```

Se declaran `i, j, n y temporal` como varaiables enteros y se usan de contadores. También se definen el array vector_estado como entero positivo, en el se almacena el texto plano a cifrar o descifrar. Del mismo modo, se declaran como entero positivo los vectores y matrices `clave, matriz_claves, vector_claves, sub_claves` que se utilizan para calcular todas las claves de ronda.

Pero las declaraciones de variables más importantes son las que verdaderamente están relacionadas con CUDA, se expican a continuación:

```
unsigned int *dev_vector_estado,*dev_vector_claves;
// Reservo los vectores de estado y claves como memoria global
cudaMalloc ((void**) &dev_vector_estado,N*sizeof(unsigned int));
cudaMalloc ((void**) &dev_vector_claves,176*sizeof(unsigned int));
```

Con estas sentencias se declaran y reservan las variables que más tarde se utilizan en el device (GPU). Se reservan como memoria global con la sentencia **`cudaMalloc`**, además también se indican su nombre, así como su dimensión y tipología, en este caso ambos vectores son enteros positivos.

A continuación se procede a hallar todas las claves de ronda, despúes se introducen las claves de un modo ordenado en el array `vector_claves`. El cálculo de las subclaves no se ejecuta en el **kernel** y por lo tanto no se paraleliza, porque no se puede ya que es un proceso recurrente y por lo tanto de imposible paralelización. Este procedimiento se puede consultar en el ANEXO C o D.

Inmediatamente después, se procede a realizar los traspasos de memoria y las llamadas al kernel con las siguientes sentencias:

```
//Traspaso las variables necesarias a la la GPU
cudaMemcpy(dev_vector_estado, vector_estado, N*sizeof(unsigned int),
cudaMemcpyHostToDevice);
cudaMemcpy(dev_vector_claves, vector_claves, 176*sizeof(unsigned int),
cudaMemcpyHostToDevice);
//Invoco al KERNEL
aes_funciones_codifica_kernel<<<1,N>>>(dev_vector_estado, dev_vector_claves);
//Devuelvo lo calculado en la GPU a la CPU
cudaMemcpy(vector_estado, dev_vector_estado, N*sizeof(unsigned int),
cudaMemcpyDeviceToHost);
```

A través de la instrucción **`cudaMemcpy`** se pueden realizar traspasos de memoria, para ello se debe indicar la dirección donde se quiere copiar la información, donde actualmente están situados los datos, el tamaño y tipo de datos y finalmente si se está copiando de la CPU a la GPU o al contrario.

De este modo se usa **`cudaMemcpyHostToDevice`** para copiar de la CPU a la GPU y **`cudaMemcpyDeviceToHost`** para transferir memoria de la GPU a la CPU. Se recueda que host hace referencia al uso de la CPU y device o dispositivo a la GPU.

En esta sección también se llama o invoca al **kernel**, aunque su explicación y definición se desarrolla en otro apartado. Para invocarlo solo hay que llamarlo e indicar las dimensiones sobre las cuales se va a ejecutar, en este caso se implementa en un bloque de N hilos o

threads, donde N es el número de componentes que forman el texto plano en cada ensayo. Además también se le deben indicar los parámetros necesarios para su ejecución.

```
cudaFree(dev_vector_estado);
cudaFree(dev_vector_claves);
```

Después de los correspondientes traspasos de memoria se debe devolver lo calculado a la CPU, además se debe liberar memoria mediante la instrucción **`cudaFree`** `()` indicando en el paréntesis la variable que se quiere liberar.

La única diferencia que existe en el main entre el proceso de cifrado y el de descifrado es la llamada al kernel. En el caso del proceso de cifrado se acaba de comentar dicha llamada, pero para el proceso de descifrado la llamada es muy similar pero con otro nombre, ya que corresponde al proceso inverso de cifrado.

```
aes_funciones_descodifica_kernel<<<1,N>>>(dev_vector_estado, dev_vector_claves);
```

6.5.3. Kernel

El **kernel** es la función que se ejecuta en paralelo en la GPU, el kernel se ejecuta en N hilos diferentes a la vez. Así pues en la programación del kernel es donde se deben aplicar los conocimientos en CUDA y se debe buscar una configuración óptima para aprovechar la potencia de los hilos.

Se debe recordar que el kernel se ejecuta en el device (GPU), por lo que se paraleliza, pero se invoca en el host (CPU).

En esta sección se explica el funcionamiento para ambos kernel tanto para el proceso de cifrado como de descifrado. Ambos kernel tienen un funcionamiento muy similar, ya que cada uno utiliza cuatro funciones paralelas y los mismos pramétros. En los siguientes apartados se explica cada uno de ellos.

6.5.3.1. Funciones y kernel para cifrar

En el siguiente estracto de código se puede observar el modo en el que se declara el kernel para el proceso de cifrar:

```
__global__ void aes_funciones_codifica_kernel (unsigned int*vector_estado,
unsigned int*vector_claves)
{      int l=0,n;
       {  n=0;
       addRounkey ( vector_claves,  vector_estado, n);
        for(l=0;l<=8;l++)
       {
              subBytes (vector_estado);
              shiftRows (vector_estado);
              mixColumns (vector_estado);
              n=n+1;
              addRounkey ( vector_claves,  vector_estado, n);
        }
         subBytes (vector_estado);
         shiftRows (vector_estado);
         n=n+1;
         addRounkey ( vector_claves,  vector_estado, n);
       }
}
```

Para definir el kernel es necesario utilizar la sentencia `__global__ void`, después se le debe asignar un nombre, en este caso `aes_funciones_codifica_kernel`. Además se debe indicar los parámetros que necesita el kernel.

Como variables locales enteras se definen l y n. El parámetro l se utiliza para calcular las rondas y n para avanzar en el vector que contiene todas las claves de ronda. Además se utilizan cuatro funciones que es necesario declarar, para declarar estas funciones como se ejecutan en la GPU se debe utilizer la sentencia `__device__ void`.

A continuación se analizan estas funciones:

```
__device__ void addRounkey (unsigned int*vector_claves, unsigned int*vector_estado,int n)
{
  int i;
  int tID=threadIdx.x;
  for (i=0;i<4;i++)
    {
       if (tID < 16)
    {
       vector_estado[tID+(16*i)]=vector_estado[tID+(16*i)]^vector_claves[(16*n)+tID];
     }
       __syncthreads();
     }
     }
```

La función **addRoundKey** halla la X-OR entre el estado y la clave de ronda correspondiente. Esta función se usa tanto en el proceso directo como en el inverso, porque como ya se ha comentado la inversa de la X-OR es ella misma. La operación en addRoundKey se realiza mediante 16 hilos, de este modo cada estado se realiza a la vez, porque se paraleliza.

Además se declara la variable entera i para poder recorrer el texto plano completo. Del mismo modo la variable `tID` se utiliza para identificar los hilos. Los parámetros de la función son los arrays `vector_claves` y `vector_estado` para proceder a la transformación y `n` para indicar la correspondiente clave de ronda.

Se debe comentar el uso de la sentencia `__syncthreads()` por primera vez, con ella se sincronizan todos los hilos de un mismo bloque, es decir, un programa espera a que terminen de ejecutarse todos los hilos para poder seguir con la ejecución.

```
__device__ void subBytes(unsigned int* vector_estado)
{
       int tID=threadIdx.x;
int sbox[256] =   {
    //0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};//F
if (tID < N)
 {
```

```
        vector_estado[tID]=sbox[vector_estado[tID]];
 }
}
```

En la función **subBytes** se sustituye cada byte del estado por su equivalente en la matriz de sustitución de bytes. Esta función se pararleliza completamente mediante N hilos, es decir se utilizan tantos threads como componentes forman el texto plano. La única variable local que se define es el arrray que conforma la tabla S-BOX. El parámetro que se le pasa a la función es el vector que compone el estado. Al igual que en el caso anterior `tID` indica los distintos hilos.

```
__device__ void shiftRows (unsigned int* vector_estado)
{
  int tID=threadIdx.x;
  int i;
  unsigned int estado_auxiliar[N];
       for(i=0;i<4;i++)
{
       //Primera fila
       estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
       estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
       estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
       estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
       //Segunda fila
       estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+5];
       estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+6];
       estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+7];
       estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+4];
       //Tercera fila
       estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
       estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
       estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
       estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
       //Cuarta fila
       estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+15];
       estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+12];
       estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+13];
       estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+14];
}
       if (tID < N)
       {
              vector_estado[tID]=estado_auxiliar[tID];
       }
}
```

En el caso de la función **shiftRows** también se utiliza como parámetro el array `vector_estado`, en él se efectúan las rotaciones circulares. Esas rotaciones no se paralelizan, se realizan a través de un estado intermedio que se declara como variable local entera positiva. Lo que si se paraleliza en N hilos es la copia de los datos que componen la variable `estado_auxiliar` a la variable `vector_estado`.

Además, al igual que en otras ocasiones se utiliza la variable i para recorrer la totalidad de un vector. Como se puede comprobar, la función shiftRows no es fácilmente paralelizable.

```
__device__ void mixColumns(unsigned int* vector_estado)
{
       int i;
       int tID=threadIdx.x;
       unsigned char Tmep,Tme,time;
// xtime es un macro que devuelve el producto con modulo {1b} de {02} y el byte argumento
   #define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))
```

```
   for (i=0;i<4;i++)
   {
       if (tID < 4)
       {
        time=vector_estado[(16*i)+tID];
        Tmep = vector_estado[(16*i)+tID] ^ vector_estado[(16*i)+4+tID] ^
vector_estado[(16*i)+8+tID] ^ vector_estado[(16*i)+12+tID] ;
         Tme = vector_estado[(16*i)+tID] ^ vector_estado[(16*i)+4+tID] ;
         Tme = xtime(Tme);
         vector_estado[(16*i)+0+tID] ^= Tme ^ Tmep ;
         Tme = vector_estado[(16*i)+4+tID] ^ vector_estado[(16*i)+8+tID] ;
         Tme = xtime(Tme);
         vector_estado[(16*i)+4+tID] ^= Tme ^ Tmep ;
         Tme = vector_estado[(16*i)+8+tID] ^ vector_estado[(16*i)+12+tID] ;
         Tme = xtime(Tme);
         vector_estado[(16*i)+8+tID] ^= Tme ^ Tmep ;
         Tme = vector_estado[(16*i)+12+tID] ^ time ; Tme = xtime(Tme);
         vector_estado[(16*i)+12+tID] ^= Tme ^ Tmep ;
       }
    }
}
```

Finalmente se implementa la función mixColumns, su único parámetro es vector_estado. Se debe comentar que al igual que en el caso del programa en C, para su realización se parte del código de Eduardo Bonilla [39].

En esta función se multiplica cada columna por una matriz en el Campo de Galois. Además del indicador de hilo también se usan una serie de variables auxiliaries e i para recorrer la totalidad del texto plano. En este caso se paraleliza cada cuatro elementos, es decir, el equivalente a cada columna del estado.

6.5.3.2. Funciones y kernel para descifrar

A continuación se explican las funciones y el kernel para el proceso **inverso de cifrado**, este proceso es muy similar al del caso directo.

```
__global__ void aes_funciones_descodifica_kernel (unsigned int*vector_estado, unsigned
int*vector_claves)
{
 int n=10;
 int l;
              {
              addRounkey ( vector_claves,  vector_estado, n);
              for(l=0;l<=8;l++)
              {
                     inv_shiftRows(vector_estado);
                     inv_subBytes(vector_estado);
                     n=n-1;
                     addRounkey ( vector_claves,  vector_estado, n);
                     inv_mixColumns (vector_estado);
              }
                     inv_shiftRows(vector_estado);
                     inv_subBytes(vector_estado);
                     n=n-1;
                     addRounkey ( vector_claves,  vector_estado, n);
                     n=10;
              }
}
```

Como se puede ver, el kernel es muy parecido al del proceso inverso, pero en este caso se utilizan las funciones inversas. Por este motivo, su funcionamiento no se explica de manera tan detallada. Los parámetros del kernel son `vector_estado` y `vector_claves`. Del mismo modo l se utiliza para realizar las diferentes rondas y n para seleccionar la correspondiente clave de ronda, la única diferencia es que ahora n toma otros valores.

La función **addRoudKey** no se vuelve a explicar, porque su funcionamiento es igual que en el proceso directo. Las otras funciones sufren algunas variaciones que a continuación se explican.

```
__device__ void inv_shiftRows(unsigned int* vector_estado)
{
  int tID=threadIdx.x;
  int i;
unsigned int estado_auxiliar[N];
for(i=0;i<4;i++)
{
	//Primera fila
	estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
	estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
	estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
	estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
	//Segunda fila
	estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+7];
	estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+4];
	estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+5];
	estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+6];
	//Tercera fila
	estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
	estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
	estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
	estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
	//Cuarta fila
	estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+13];
	estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+14];
	estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+15];
	estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+12];
}
	if (tID < N)
	{
		vector_estado[tID]=estado_auxiliar[tID];
	}
}
```

La función **inv_shiftRows** tiene un funcionamiento muy similar a su equivalente en el proceso directo, pero en este caso las rotaciones se realizan hacia el lado contrario. Por lo demás, las rotaciones no se paralelizan, pero si se paraleliza el traspaso dedatos de la variable auxiliary a `vector_estado`.

```
__device__ void inv_subBytes(unsigned int* vector_estado)
{
int tID=threadIdx.x;
int isbox[256] = {
    //0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, //0
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, //1
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, //2
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, //3
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, //4
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, //5
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, //6
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, //7
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73, //8
```

```
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, //9
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, //A
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, //B
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f, //C
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, //D
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, //E
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};//F
        if (tID < N)
        {
                vector_estado[tID]=isbox[vector_estado[tID]];
        }
}
```

El funcionamiento de **inv_subBytes** es totalmente análogo al de la función subBytes, pero en este caso se utiliza la matriz inversa de sustitución. Esta función se paraleliza completamente, es decir, se utilizan tantos hilos como componentes tenga el texto plano.

```
__device__ void inv_mixColumns (unsigned int* vector_estado)
{
  int tID=threadIdx.x;
  int i;
  #define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))
int auxiliar[N], num_a,num_b,num_c,num_d;
#define Multiply(x,y) (((y & 1) * x) ^ ((y>>1 & 1) * xtime(x)) ^ ((y>>2 & 1) *
xtime(xtime(x))) ^ ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^ ((y>>4 & 1) *
xtime(xtime(xtime(xtime(x))))))
for(i=0;i<4;i++)
{
if(tID<4)
{
num_a = vector_estado[(16*i)+tID];
num_b = vector_estado[(16*i+4)+tID];
num_c = vector_estado[(16*i+8)+tID];
num_d = vector_estado[(16*i+12)+tID];
vector_estado[(16*i)+tID] = Multiply(num_a, 0x0e) ^ Multiply(num_b, 0x0b) ^
Multiply(num_c, 0x0d) ^ Multiply(num_d, 0x09);
vector_estado[(16*i+4)+tID] = Multiply(num_a, 0x09) ^ Multiply(num_b, 0x0e) ^
Multiply(num_c, 0x0b) ^ Multiply(num_d, 0x0d);
vector_estado[(16*i+8)+tID] = Multiply(num_a, 0x0d) ^ Multiply(num_b, 0x09) ^
Multiply(num_c, 0x0e) ^ Multiply(num_d, 0x0b);
vector_estado[(16*i+12)+tID] = Multiply(num_a, 0x0b) ^ Multiply(num_b, 0x0d) ^
Multiply(num_c, 0x09) ^ Multiply(num_d, 0x0e);
}
}
__syncthreads();
if (tID < N)
{
  auxiliar[tID]=vector_estado[tID]/0x100;
  vector_estado[tID]=vector_estado[tID]-auxiliar[tID]*0x100;
}
}
```

Para realizar esta función también se parte del código del PFC de Eduardo Bonilla [39]. En la función **inv_mixColumns** se multiplica cada columna por una matriz en el Campo de Galois, esta matriz es diferente a la de la función mix_Columns, pero están relacionadas.

Esta función es difícil de paralelizar, así que solamente se utilizan cuatro hilos, de esta manera se paralelizan las operaciones por columnas. Lo que si se paraleliza completamente es la operación para que todas las componentes que forman el estado tengan solo dos componentes, este proceso si se paraleliza en N threads.

6.5.4. Cálculo de tiempos

El cálculo del tiempo de ejecución se realiza de dos modos distintos, con el uso de dos librerías diferentes. En un primer momento se utiliza la librería `<time.h>` de la misma manera que se utiliza en el lenguaje C, pero también se utiliza la librería `<windows.h>` para buscar mayor precisión.

El procedimiento con `<time.h>` ya se comentó anteriormente. Así pues se utilizan dos variables tipo `clock_t`, una para iniciar el evento y otra para terminarlo. Del mismo modo también es necesario otra variable tipo `double` para realizar la resta de ambos eventos [40]. En el caso de la implementación en CUDA, el tiempo que se registra con esta librería es desde que se realiza la primera trasferencia de memoria hasta la última, incluyendo la llamada al kernel. Este procedimiento se puede ver a continuación:

```
//Defino las variables necesarias
clock_t begin, end;

double time_spent;

//Inicio el contador de tiempo
begin = clock();
//Traspaso las variables necesarias a la la GPU
cudaMemcpy(dev_vector_estado, vector_estado, N*sizeof(unsigned int),
cudaMemcpyHostToDevice);
cudaMemcpy(dev_vector_claves, vector_claves, 176*sizeof(unsigned int),
cudaMemcpyHostToDevice);
//Invoco al KERNEL
aes_funciones_codifica_kernel<<<1,N>>>(dev_vector_estado, dev_vector_claves);
//Devuelvo lo calculado en la GPU a la CPU
cudaMemcpy(vector_estado, dev_vector_estado, N*sizeof(unsigned int),
cudaMemcpyDeviceToHost);
//Finalizo el contador de tiempo
end = clock();
//Calculo el tiempo de ejecución
time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

Además también se utiliza la sentencia `CLOCKS_PER_SEC` que se refiere al número de pulsos de reloj por segundo.

Se mide justo ese tiempo porque es el equivalente al que se mide con la misma función en el programa escrito en el lenguaje en C. Finalmente se debe decir que con este procedimiento se obtiene el tiempo en segundos, pero con una precisión de milésimas. El problema es la escasa exactitud de este método, especialmente para mediciones pequeñas debido a la inestabilidad del método.

El otro procedimiento es con la librería `<windows.h>`, se realiza mediante una función que se debe programar previamente, esta función calcula la resta de dos eventos con gran precision. También hacen falta dos variables tipo `LARGE_INTEGER` para registrar los momentos de principio y fin de los eventos y una de tipo `double` para almacenar la resta. Para utilizar el contador de alta resolución se utiliza la sentencia `QueryPerformanceCounter` [53].

```
//Función devuelve la resta de dos eventos
double performancecounter_diff(LARGE_INTEGER *a, LARGE_INTEGER *b)
{
  LARGE_INTEGER freq;
  QueryPerformanceFrequency(&freq);
  return (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
}
```

```
LARGE_INTEGER t_ini, t_fin;

double secs

QueryPerformanceCounter(&t_ini);
...
Realizo la ejecución del programa
...
QueryPerformanceCounter(&t_fin);
secs = performancecounter_diff(&t_fin, &t_ini);
printf("%.16g milesimas total \n", secs * 1000.0);
```

Con este nuevo procedimiento se pretende evaluar el tiempo en cada uno de los traspasos de memoria y en la llamada al kernel. La librería `<windows.h>` mide el tiempo en milésimas de segundo, pero con un gran número de decimales, por lo que esta medida tiene más exactitud y sirve de ayuda para regular el tiempo que tarda en ejecutarse cada instrucción.

6.5.5. Cifrado y descifrado con otros tamaños de clave

Al igual que en el caso del programa en C, también se realizan ensayos con diferentes tamaños de texto plano. Así pues se deben realizar modificaciones en los códigos de los ANEXOS C y D.

Solamente hay que cambiar el valor de N que se define con la sentencia `#define N` al principio del programa, con esta instrucción automáticamente se define el tamaño del array `vector_estado[N]`.

Las funciones sufren variaciones en los bucles para recorrer siempre la totalidad de componentes de la variable `vector_estado`.

Capítulo 7. Resultados

7.1. Introducción

En el presente capítulo se muestran los tiempos de ejecución para todos los resultados obtenidos en las diferentes implementaciones. Estos estudios se realizan tanto para cifrar como para descifrar. Estas implementaciones se diferencian principalmente en el tamaño del texto plano y en la memoria que se utiliza para realizar los cálculos.

De este modo, se realizan estudios para 12 tamaños de texto plano, que abarcar valores desde las 16 componentes o un estado de AES (equivalente a 128 bits) hasta las 1024 componentes o 64 estados (equivalente a 8192 bits).

Se debe comentar que los estudios solo llegan hasta las 1024 componentes. Esto se debe a las características de la GPU utilizada, ya que el número máximo de hilos por bloque es 1024, este dato aparece en las características de la tarjeta (figura 4.1). Además en el caso de este trabajo se utiliza una disposición de un único bloque con tantos hilos como componentes tenga el texto plano, de esta manera cada componente del estado se opera en un hilo diferente.

El cálculo de tiempos en las diferentes implementaciones se realiza de dos modos diferentes, es decir, a través de las dos librerías que ya se han explicado en capítulos anteriores.

Además como el tamaño del texto plano no es muy grande, se han utilizado bucles para realizar los procesos de cifrado un mayor número de veces.

Con estas premisas se obtienen diferentes tipos de ensayos con sus correspondientes resultados. Estos resultados se muestran en las tablas que aparecen en el siguiente punto:

7.2. Resultados

En este apartado se explican todos los resultados obtenidos en los diferentes ensayos tanto para cifrar como para descifrar en las diferentes implementaciones del algoritmo AES. Del mismo modo también se procede a tratar estadísticamente los datos.

7.2.1. Resultados en el caso de cifrar

En primer lugar se muestran los resultados obtenidos para el algoritmo AES en modo cifrador. De esta manera, se analizan los resultados obtenidos utilizando tanto la memoria global como la compartida para el caso de cifrar. Además se utilizan diferentes tamaños de texto plano, de este modo se obtienen múltiples resultados.

El código con la memoria global es el que se explica en el capítulo anterior. El uso de la memoria compartida se diferencia de ese mismo código únicamente dentro del kernel, ya que se declaran unas variables de tipo compartido (`__shared__`) en las que se vuelcan los datos transferidos de la CPU para aprovechar la rapidez de la memoria compartida. Este concepto se puede entender mejor con las siguientes líneas de código.

```
__shared__ unsigned int v_estado[N], v_claves[176];
```

En primer lugar declaro las variables mencionadas como compartidas (`v_estado` y `v_claves`), porque este tipo de memoria es más rápido, en estas variables copio los datos procedentes de la CPU.

```
// Paso los datos de la memoria global a la compartida
v_claves[threadIdx.x] = vector_claves[threadIdx.x];

if(threadIdx.x<N)

{
```

```
v_estado[threadIdx.x] = vector_estado[threadIdx.x];
...
Resto de líneas de código operando con las variables compartidas
...
// Vuelvo a pasar los datos de la memoria compartida a la global

vector_estado[threadIdx.x] = v_estado[threadIdx.x];
}
vector_claves[threadIdx.x] = v_claves[threadIdx.x];
```

Para concluir este proceso, se devuelven los datos de las variables compartidas a las variables globales y finalmente se devuelven a la CPU.

Respecto a este primer ensayo, se debe comentar que todas las mediciones se realizan a través de la librería time.h, las unidades de esta medida son segundos. En el caso de medidas pequeñas este método es bastante impreciso, por este motivo se han utilizado una serie de bucles para conseguir que el programa tarde más en ejecutarse y muestre un resultado valido.

7.2.1.1. Cálculo del Ratio

Para poder comparar los resultados obtenidos en CUDA y C, se utiliza un ratio. Este ratio se expresa de la siguiente manera:

$$Ratio = \frac{t_{CUDA}}{t_C}$$

Donde t_{CUDA} es el tiempo obtenido en las implementaciones en CUDA y t_C el calculado mediante C.

Este primer ensayo se realiza a través de la librería time.h. Esta librería ya se analizó anteriormente y se comentó que no era muy exacta, especialmente para valores pequeños.

Debido a la dispersión en la medida del tiempo de ejecución, se calcula la media de los diferentes ratios obtenidos en los distintos ensayos, con el fin de conseguir un resultado estable para poder representarlo. Además se debe decir que como para valores pequeños los resultados no son fiables, se desprecían esos datos en el cálculo de la media. Finalmente esos valores promedios son los que se representan para observar la variación que sufre el ratio con los diferentes tamaños de texto plano.

En la siguiente tabla se ven todos los resultados obtenidos en los casos de memoria global y compartida, para doce tamaños distintos de texto plano, así como sus correspondientes promedios.

En la siguiente tabla (tabla 7.1) N representa el tamaño del texto plano, es decir, el número de componentes expresadas en bytes. La columna t_Cuda (s) es el tiempo obtenido mediante las implementaciones realizadas en CUDA y medido en segundos, mientras que t_C (s) es su equivalente en el lenguaje C. La variable Ratio muestra la proporción que existe entre ambos tiempos, del modo que se representa en la fórmula anterior y por lo tanto es adimensional. Finalmente la columna Media representa los promedios de los diferentes ratios obtenidos en función de la memoria y tamaño de texto plano.

Tabla 7.1 Resultados para codificar con Memoria Global y Compartida

CODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
16	1	0	0	-	34,623	0	0	-	31,793
	100	0,015	0	-		0,016	0	-	
	1000	0,203	0,015	13,533		0,203	0,015	13,533	
	10000	1,997	0,062	32,209		2,059	0,062	33,209	
	100000	20,030	0,577	34,714		20,482	0,577	35,497	
	200000	40,076	1,138	35,216		40,950	1,138	35,984	
	500000	100,214	2,823	35,499		102,398	2,823	36,272	
	1000000	200,335	5,647	35,476		204,781	5,647	36,263	
32	1	0	0	-	27,587	0	0	-	25,912
	100	0,015	0	-		0,031	0	-	
	1000	0,281	0	-		0,265	0	-	
	10000	2,901	0,109	26,614		2,710	0,109	24,862	
	100000	28,798	1,029	27,986		27,144	1,029	26,379	
	200000	57,735	2,059	28,040		54,288	2,059	26,366	
	500000	144,362	5,210	27,708		135,689	5,210	26,043	
64	1	0	0	-	25,284	0	0	-	24,957
	20	0,016	0	-		0,016	0	-	
	40	0,016	0	-		0,016	0	-	
	200	0,078	0,015	5,200		0,078	0,015	5,200	
	500	0,203	0,015	13,533		0,187	0,015	12,466	
	1000	0,390	0,031	12,580		0,390	0,031	12,580	
	2000	0,795	0,031	25,645		0,765	0,031	24,677	
	5000	1,950	0,078	25,000		1,935	0,078	24,807	
	10000	3,900	0,140	27,857		3,853	0,140	27,521	
	100000	38,891	1,653	23,527		38,501	1,653	23,291	
	200000	77,969	3,291	23,691		77,002	3,291	23,397	
	300000	116,548	4,274	27,269		115,518	4,274	27,028	
	500000	194,252	7,129	27,248		192,520	7,129	27,005	
	1000000	389,876	15,444	25,244		385,055	15,444	24,932	
	2000000	777,517	32,838	23,677		770,080	32,838	23,450	
	2500000	972,050	41,043	23,683		962,791	41,043	23,458	
96	1	0	0	-	21,339	0	0	-	19,416
	100	0,063	0	-		0,047	0	-	
	1000	0,561	0,031	18,096		0,499	0,031	16,096	
	10000	5,507	0,265	20,781		5,008	0,265	18,898	
	100000	54,943	2,277	24,129		50,013	2,277	21,964	
	200000	109,871	5,662	19,404		100,027	5,662	17,666	
	500000	275,075	13,072	21,043		250,169	13,072	19,137	

CODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
192	1	0	0	-	20,911	0	0	-	18,812
	100	0,094	0	-		0,078	0	-	
	1000	0,951	0,046	20,673		0,858	0,046	18,652	
	10000	9,563	0,499	19,164		8,596	0,499	17,226	
	100000	95,532	4,368	21,870		85,909	4,368	19,667	
	200000	191,272	8,720	21,934		171,818	8,720	19,703	
288	1	0	0	-	19,936	0	0	-	17,875
	100	0,140	0	-		0,125	0	-	
	1000	1,372	0,062	22,129		1,232	0,062	19,870	
	10000	13,743	0,733	18,748		12,308	0,733	16,791	
	100000	137,374	7,256	18,932		123,100	7,256	16,965	
384	1	0	0	-	19,283	0	0	-	17,201
	100	0,188	0	-		0,172	0	-	
	1000	1,810	0,109	16,605		1,622	0,109	14,880	
	10000	18,174	0,936	19,416		16,209	0,936	17,317	
	100000	181,617	9,484	19,149		162,038	9,484	17,085	
512	1	0	0	-	19,228	0	0	-	17,292
	2	0	0	-		0	0	-	
	5	0,016	0	-		0,015	0	-	
	10	0,031	0	-		0,16	0	-	
	20	0,047	0,015	3,133		0,046	0,015	3,067	
	50	0,125	0	-		0,109	0	-	
	100	0,250	0,015	16,667		0,234	0,015	15,600	
	1000	2,465	0,124	19,879		2,200	0,124	17,741	
	10000	24,663	1,248	19,762		22,012	1,248	17,637	
	100000	246,683	12,308	20,042		220,001	12,308	17,874	
	200000	494,661	24,991	19,793		439,983	24,991	17,605	
640	1	0	0	-	20,808	0	0	-	18,907
	2	0	0	-		0	0	-	
	5	0,015	0	-		0,015	0	-	
	10	0,031	0	-		0,031	0	-	
	20	0,062	0	-		0,062	0	-	
	50	0,156	0	-		0,141	0	-	
	100	0,312	0,015	20,800		0,297	0,015	19,800	
	1000	3,214	0,156	20,602		2,886	0,156	18,500	
	10000	32,011	1,544	20,732		28,751	1,544	18,621	
	100000	320,425	15,288	20,959		287,571	15,288	18,810	
	200000	640,781	30,586	20,950		575,157	30,586	18,804	

CODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
768	1	0	0	-	20,247	0	0	-	17,872
	2	0,016	0	-		0,015	0	-	
	5	0,016	0	-		0,015	0	-	
	10	0,032	0,015	2,133		0,032	0,015	2,1333	
	20	0,078	0,031	2,516		0,063	0,031	2,0322	
	50	0,187	0,015	12,466		0,187	0,015	12,466	
	100	0,405	0,031	13,064		0,359	0,031	11,580	
	1000	4,009	0,187	21,438		3,620	0,187	19,358	
	10000	40,060	1,825	21,950		36,223	1,825	19,848	
	20000	88,137	3,790	23,255		72,462	3,790	19,119	
	30000	120,214	5,584	21,528		108,654	5,584	19,458	
896	1	0,016	0	-	21,995	0,015	0	-	20,090
	2	0,016	0	-		0,016	0	-	
	5	0,031	0	-		0,031	0	-	
	10	0,047	0,015	3,133		0,047	0,015	3,133	
	20	0,109	0,015	7,266		0,093	0,015	6,200	
	50	0,249	0	-		0,234	0	-	
	100	0,515	0,031	16,612		0,468	0,031	15,096	
	1000	5,023	0,218	23,041		4,602	0,218	21,110	
	10000	50,310	2,137	23,542		45,942	2,137	21,498	
	20000	100,604	4,274	23,538		91,900	4,274	21,502	
	30000	150,805	6,489	23,240		137,857	6,489	21,244	
1024	1	0,015	0	-	23,312	0	0	-	21,420
	5	0,031	0	-		0,031	0	-	
	10	0,062	0	-		0,062	0	-	
	50	0,312	0,015	20,800		0,281	0,015	18,733	
	100	0,624	0,031	20,129		0,577	0,031	18,612	
	1000	6,240	0,265	23,547		5,756	0,265	21,720	
	10000	62,322	2,511	24,819		57,408	2,511	22,862	
	20000	124,582	5,020	24,817		114,770	5,020	22,862	
	30000	186,867	7,254	25,760		172,143	7,254	23,730	

Como se puede ver el tiempo en C siempre es mucho menor que el de CUDA. Como ya se ha comentado anteriormente, el ratio representa cuantas veces es más grande el tiempo en CUDA, de igual modo se puede ver como en general este ratio disminuye a medida que aumenta el tamaño del texto plano. Aunque también se produce una anomalía que se comenta más tarde.

En el siguiente gráfico se puede ver esa tendencia que tiene el ratio al aumentar el tamaño del texto plano en el caso de cifrar un mensaje, tanto para la memoria global como para la compartida.

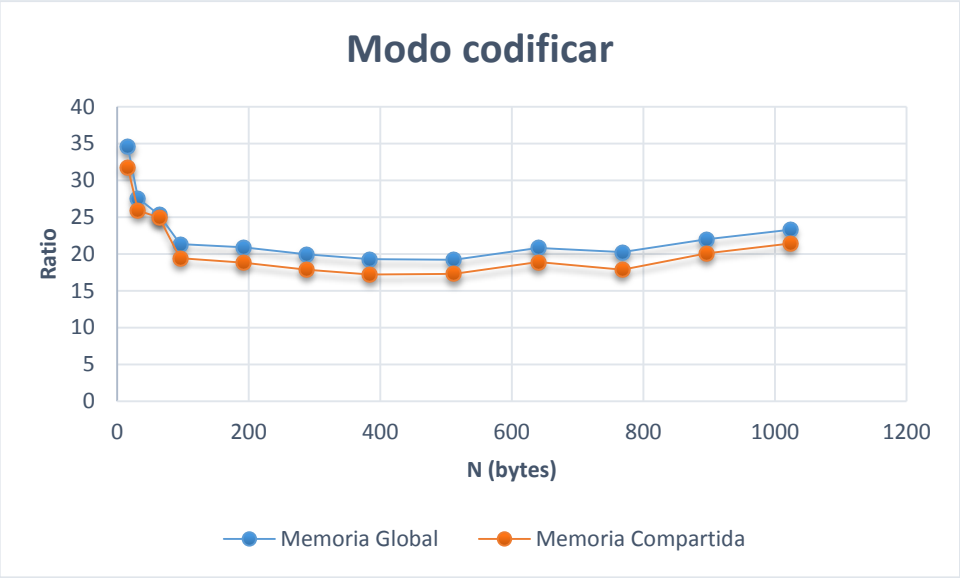

Figura 7.1 Representación del ratio para codificar con Memoria Global y Compartida

A la vista de los resultados de la gráfica se puede comprobar que al aumentar el tamaño del texto plano el ratio disminuye y que utilizar la memoria compartida implica también una mejoría en ese ratio, ya que el ratio es siempre menor para el caso de la memoria compartida. A continuación analizamos que porcentaje supone esa mejora, para ello se utiliza la siguiente fórmula:

$$\%\, Mejora = \frac{Ratio_{M.Global} - Ratio_{M.Compartida}}{Ratio_{M.Global}} \cdot 100$$

Para el cálculo del porcentaje de la mejora se utilizan los ratios medios de los distintos tamaños que aparecen en la tabla anterior. De esta manera se obtienen los siguientes resultados:

Tabla 7.2 Mejora con la Memoria Compartida para codificar

Nº Componentes o bytes	%Mejora
16	8,172
32	6,070
64	1,294
96	9,011
192	10,034
288	10,337
384	10,796
512	10,072
640	9,138
768	11,727
896	8,659
1024	8,114
Media	**8,619**

Como se puede comprobar al transferir los datos y usar la memoria compartida se obtiene una cierta reducción. En concreto se obtiene de media una reducción de **8.62%** en el ratio que compara ambas implementaciones.

7.2.1.2. Velocidad de procesamiento

De la misma manera también se puede representar la velocidad de procesamiento para ambas memorias. Esta velocidad se puede calcular a través de la siguiente fórmula:

$$Velocidad\ de\ procesamento\ \left(\frac{MB}{s}\right) = \frac{N \cdot Nº\ iteraciones}{t_{CUDA} \cdot 2^{20}}$$

Con esta operación se obtiene los siguientes datos:

Tabla 7.3 Velocidad de procesamiento para codificar con Memoria Global, Compartida y CPU

N (bytes)	Vel. Procesamiento (MB/s) Memoria Global	Vel. Procesamiento (MB/s) Memoria Compartida	Vel. Procesamiento(MB/s) CPU
16	0,076	0,074	2,368
32	0,106	0,112	2,914
64	0,155	0,158	3,956
96	0,165	0,183	3,432
192	0,192	0,217	4,010
288	0,199	0,222	3,987
384	0,200	0,222	3,711
512	0,198	0,218	3,796
640	0,194	0,207	3,983
768	0,192	0,212	3,932
896	0,169	0,184	3,966
1024	0,156	0,167	3,651

La velocidad de procesamiento en la CPU es mucho mayor que en CUDA. En el caso de la velocidad de procesamiento en CUDA, se observa que la velocidad aumenta con el tamaño del texto plano. Además en esta ocasión la velocidad de procesamiento es mayor en el caso de la memoria compartida, lo que concuerda totalmente con los resultados anteriores.

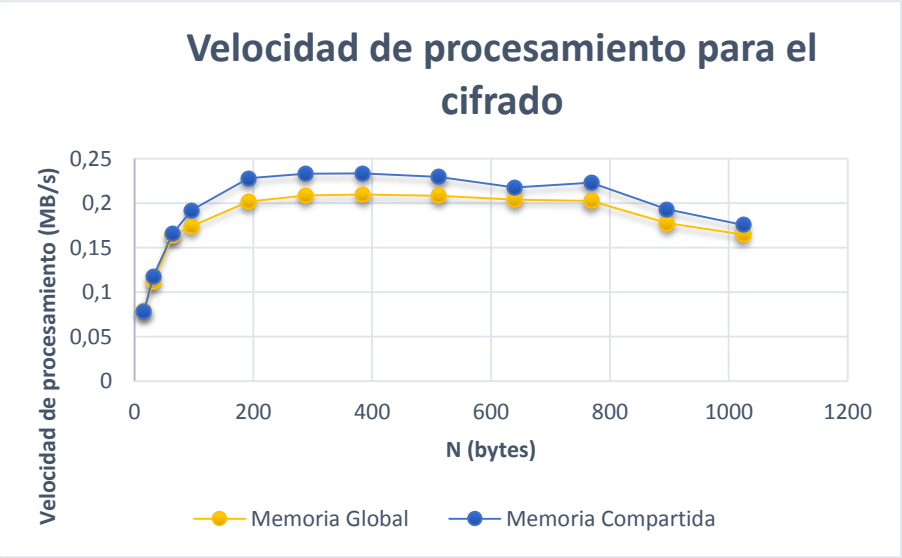

Figura 7.2 Velocidad de procesamiento para cifrar con Memoria Global y Compartida

Como ya se ha comentado, el procedimiento mejora de manera general al aumentar el tamaño de texto plano, ya que el ratio disminuye y la velocidad de procesamiento aumenta. El problema surge a partir de un cierto tamaño de texto plano, ya que en ese momento los resultados comienzan a empeorar, las posibles causas de esto se analizan posteriormente en las conclusiones.

7.2.2. Resultados en el caso de descifrar

En el caso del proceso de descifrado se opera de un modo totalmente análogo al procedimiento anterior, es decir, se calcula el ratio y la velocidad de procesamiento con la memoria global y la compartida.

Por este motivo, el procedimiento no se vuelve a explicar de un modo detallado ya que las operaciones se realizan de la misma manera, la única diferencia son los resultados obtenidos.

Así pues las mediciones se realizan con la misma librería y los cálculos se efectúan con las mismas fórmulas.

7.2.2.1. Cálculo del Ratio

En la siguiente tabla adjunta se pueden ver los tiempos de ejecución así como los ratios y sus medias para el modo descodificar con la Memoria Global y la Compartida.

Tabla 7.4 Resultados para descodificar con Memoria Global y Compartida

DESCODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
16	1	0	0	-	10,566	0	0	-	10,331
	100	0,016	0	-		0,031	0	-	
	1000	0,250	0,031	8,064		0,250	0,031	8,064	
	10000	2,496	0,249	10,024		2,433	0,249	9,771	
	100000	24,913	2,230	11,171		24,274	2,230	10,885	
	200000	49,842	4,336	11,494		48,563	4,336	11,199	
	500000	124,567	10,873	11,456		121,384	10,873	11,163	
	1000000	249,179	22,270	11,189		242,784	22,270	10,901	
32	1	0	0	-	9,048	0	0	-	8,791
	100	0,031	0,015	2,066667		0,031	0,015	2,066667	
	1000	0,39	0,046	8,478261		0,374	0,046	8,130435	
	10000	3,822	0,405	9,437037		3,728	0,405	9,204938	
	100000	38,189	4,071	9,380742		37,206	4,071	9,139278	
	200000	76,378	8,564	8,918496		74,41	8,564	8,688697	
	500000	190,959	21,153	9,027514		186,014	21,153	8,793741	

DESCODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
64	1	0	0	-	7,373	0	0	-	7,418
	10	0	0	-		0	0	-	
	15	0	0	-		0,015	0	-	
	20	0,015	0	-		0	0	-	
	40	0,015	0	-		0,016	0	-	
	100	0,046	0,015	3,066		0,062	0,015	4,133	
	200	0,109	0,015	7,266		0,109	0,015	7,266	
	500	0,281	0,046	6,108		0,280	0,046	6,086	
	1000	0,561	0,078	7,192		0,562	0,078	7,205	
	2000	1,139	0,140	8,135		1,155	0,140	8,250	
	5000	2,840	0,421	6,745		2,854	0,421	6,779	
	10000	5,678	0,748	7,590		5,725	0,748	7,653	
	100000	56,816	7,316	7,765		57,206	7,316	7,819	
	200000	113,459	15,927	7,123		114,426	15,927	7,184	
	300000	170,477	22,245	7,663		171,631	22,245	7,715	
	500000	283,733	36,566	7,759		286,026	36,566	7,822	
	1000000	567,326	73,132	7,757		572,071	73,132	7,822	
96	1	0	0	-	6,779	0	0	-	6,573
	100	0,078	0	-		0,093	0	-	
	1000	0,812	0,140	5,800		0,764	0,140	5,457	
	10000	7,987	1,248	6,399		7,753	1,248	6,212	
	100000	79,919	11,887	6,723		77,454	11,887	6,515	
	200000	159,822	23,93	6,678		154,921	23,930	6,473	
	500000	399,563	54,615	7,315		387,255	54,615	7,090	
192	1	0	0	-	6,424	0	0	-	6,211
	100	0,140	0,015	9,333		0,140	0,015	9,333	
	1000	1,435	0,234	6,132		1,388	0,234	5,931	
	10000	14,352	2,277	6,303		13,868	2,277	6,090	
	100000	143,317	21,590	6,638		138,606	21,590	6,419	
	200000	286,650	43,274	6,624		277,197	43,274	6,405	
288	1	0	0	-	6,356	0	0	-	6,216
	100	0,203	0,031	6,548		0,203	0,031	6,548	
	1000	2,091	0,343	6,096		2,028	0,343	5,912	
	10000	20,826	3,291	6,328		20,218	3,291	6,143	
	100000	208,323	32,276	6,454		202,067	32,276	6,260	

DESCODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
384	1	0	0	-	6,206	0	0	-	6,034
	100	0,281	0,031	9,064		0,281	0,031	9,064	
	1000	2,762	0,452	6,110		2,684	0,452	5,938	
	10000	27,581	4,446	6,203		26,817	4,446	6,031	
	100000	275,762	43,726	6,306		268,212	43,726	6,133	
512	1	0	0	-	6,289	0	0	-	6,149
	2	0	0	-		0,016	0	-	
	5	0,016	0	-		0,016	0	-	
	10	0,047	0,015	3,133		0,031	0,015	2,066	
	20	0,062	0	-		0,078	0	-	
	50	0,187	0,031	6,032		0,187	0,031	6,032	
	100	0,374	0,062	6,032		0,375	0,062	6,048	
	1000	3,791	0,592	6,403		3,651	0,592	6,167	
	10000	37,908	5,912	6,412		36,519	5,912	6,177	
	100000	379,158	57,75	6,565		365,01	57,75	6,320	
640	1	0	0	-	6,809	0	0	-	6,600
	2	0	0	-		0	0	-	
	5	0,015	0	-		0,016	0	-	
	10	0,047	0,015	3,133		0,047	0,015	3,133	
	20	0,094	0,015	6,266		0,094	0,015	6,266	
	50	0,249	0,031	8,03		0,234	0,031	7,548	
	100	0,500	0,078	6,410		0,483	0,078	6,192	
	1000	4,946	0,733	6,747		4,774	0,733	6,512	
	10000	48,984	7,378	6,639		47,799	7,378	6,478	
	100000	489,21	72,337	6,762		477,844	72,337	6,605	
768	1	0	0	-	6,480	0	0	-	6,342
	2	0,015	0	-		0,016	0	-	
	5	0,031	0	-		0,031	0	-	
	10	0,062	0,015	4,133		0,062	0,015	4,133	
	20	0,125	0,031	4,032		0,125	0,031	4,032	
	50	0,312	0,046	6,782		0,297	0,046	6,456	
	100	0,608	0,093	6,537		0,608	0,093	6,537	
	1000	6,193	0,920	6,731		6,053	0,920	6,579	
	10000	61,948	8,751	7,078		60,512	8,751	6,914	
	20000	123,911	17,503	7,079		121,071	17,503	6,917	
	30000	185,875	26,098	7,122		181,584	26,098	6,957	

DESCODIFICAR									
N (tamaño texto plano)	Número Iteraciones	Memoria Global				Memoria Compartida			
		t_Cuda (s)	t_C (s)	Ratio	Media	t_Cuda (s)	t_C (s)	Ratio	Media
896	1	0	0	-	7,181093	0	0	-	7,076190534
	2	0,016	0	-		0,015	0	-	
	5	0,031	0	-		0,046	0	-	
	10	0,078	0,015	5,2		0,078	0,015	5,2	
	20	0,156	0,031	5,032258		0,156	0,031	5,032258	
	50	0,374	0,046	8,130435		0,374	0,046	8,130435	
	100	0,764	0,109	7,009174		0,749	0,109	6,87156	
	1000	7,613	1,029	7,398445		7,456	1,029	7,24587	
	10000	76,082	10,093	7,538096		74,568	10,093	7,388091	
	20000	152,1	20,092	7,570177		149,152	20,092	7,423452	
	30000	228,135	30,061	7,589069		223,704	30,061	7,441669	
1024	1	0,016	0	-	7,600889	0,015	0	-	7,614684832
	5	0,046	0,015	3,066667		0,046	0,015	3,066667	
	10	0,078	0,015	5,2		0,093	0,015	6,2	
	50	0,468	0,062	7,548387		0,452	0,062	7,290323	
	100	0,92	0,109	8,440367		0,905	0,109	8,302752	
	1000	9,172	1,154	7,948007		9,032	1,154	7,82669	
	10000	91,822	11,45	8,019389		90,34	11,45	7,889956	
	20000	183,659	22,869	8,030915		180,711	22,869	7,902007	
	30000	275,466	34,351	8,019155		271,066	34,351	7,891066	

Si representamos la media de los ratios para descodificar se obtiene:

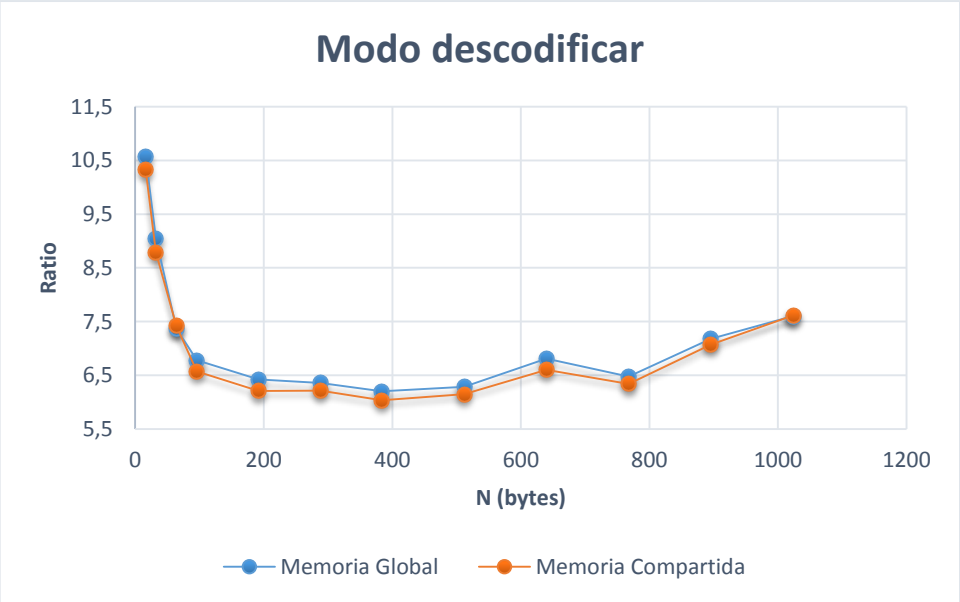

Figura 7.3 Representación del ratio para descodificar con Memoria Global y Compartida

Al igual que en el caso de cifrar se produce una mejora al utilizar la memoria compartida, aunque esta diferencia supone un menor porcentaje que en el caso actual. A continuación se calcula ese porcentaje de mejora.

Tabla 7.5 Mejora con la Memoria Compartida para descodificar

Nº Componentes o bytes	%Mejora
16	2,230
32	2,840
64	-0,610
96	3,042
192	3,307
288	2,211
384	2,776
512	2,226
640	3,069
768	2,136
896	1,460
1024	-0,181
Media	**2,042**

En el caso del descifrado la mejora solamente supone un **2.04%** pero aun así es mejor utilizar la memoria compartida. Como se puede comprobar casi en la totalidad de los datos la memoria global es más rápida, pero en algunos puntos los resultados son muy parecidos entre ambas memorias, pero esto se debe a la imprecisión de la medida.

Además vuelve a ocurrir lo mismo que en el cifrado, a partir de un cierto punto, los datos comienzan a empeorar, este hecho se comenta en el próximo capítulo.

7.2.2.2. Velocidad de procesamiento

En esta ocasión también se procede de un modo análogo al cálculo del mismo parámetro para el proceso de cifrado. Es decir se utiliza la misma fórmula y procedimiento que en el caso del cifrado.

Así pues, los datos obtenidos para la velocidad de procesamiento en el caso del descifrado con AES son:

Tabla 7.6 Velocidad de procesamiento para descodificar con Memoria Global, Compartida y CPU

N (bytes)	Vel. Procesamiento (MB/s) Memoria Global	Vel. Procesamiento (MB/s) Memoria Compartida	Vel. Procesamiento (MB/s) CPU
16	0,061	0,062	0,646
32	0,079	0,081	0,734
64	0,111	0,102	0,796
96	0,114	0,118	0,593
192	0,128	0,131	0,820
288	0,132	0,135	0,828
384	0,132	0,134	0,823
512	0,115	0,112	0,656
640	0,120	0,121	0,657
768	0,118	0,120	0,827
896	0,111	0,113	0,845
1024	0,108	0,107	0,848

A la vista de los resultados la diferencia entre la velocidad de procesamiento con memoria global o compartida es mínima, aunque es cierto que en general es mayor la velocidad de la memoria compartida. Además también se puede comprobar que la velocidad en C es mucho mayor que cualquiera de las memorias.

Los datos de las memorias global y compartida se pueden ver en la siguiente figura. Se debe destacar que debido a la poca diferencia que existe entre los resultados, los puntos casi son coincidentes, es decir, no aparece tanta diferencia entre ellos como ocurría en el punto anterior en el que se analizaba el proceso de cifrado.

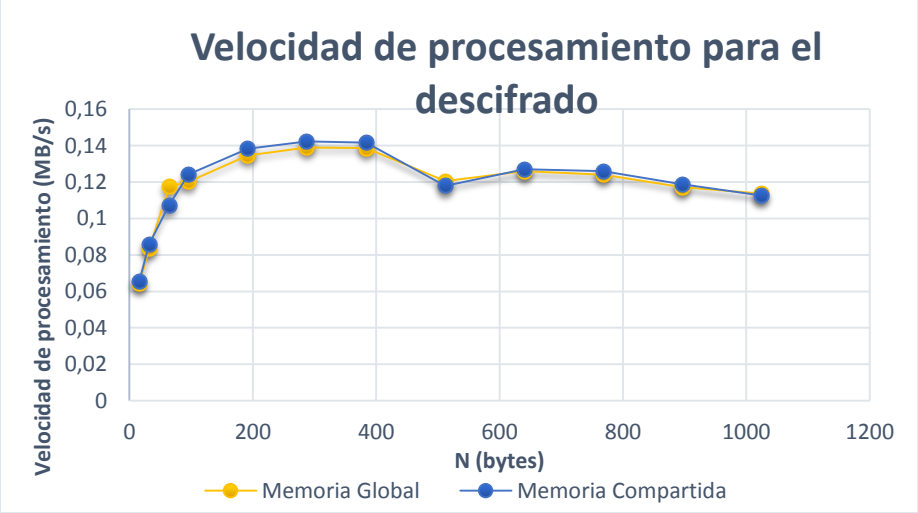

Figura 7.4 . Velocidad de procesamiento para descifrar con Memoria Global y Compartida

7.2.3. Desglose datos en el proceso de cifrado

Para comprender mejor el tiempo que consume cada instrucción en CUDA, es decir, los traspasos de memoria y el kernel, se utiliza otra biblioteca para realizar un nuevo ensayo. Así pues se obtienen con la librería windows.h el tiempo en milésimas de segundos para los traspasos de memoria y las llamadas al kernel, el funcionamiento de este procedimiento se explica en el capítulo referente a CUDA.

Este nuevo ensayo se realiza para el caso de cifrar con Memoria Global y Compartida, los resultados se muestran en la tabla adjunta.

Donde N representa el tamaño del texto plano, t_C el tiempo que tarda en ejecutarse en C, t_Tot el tiempo total que tarda en CUDA, t_h_d es el tiempo de las trasferencias host to device, t_ker el tiempo que tarda el ejecutarse la llamada al kernel y finalmente t_d_h representa la última transferencia de memoria entre la GPU y la CPU.

Tabla 7.7 Desglose datos tiempos de transferencias de memoria y kernel

CODIFICAR										
N	Nº Iter.	t_C (ms)	Memoria Global (ms)				Memoria Compartida (ms)			
			t_Tot	t_h_d	t_ker	t_d_h	t_Tot	t_h_d	t_ker	t_d_h
16	1	0,007	0,7626	0,07642	0,3917	0,2922	0,7319	0,07603	0,3744	0,27994
	100	0,571	22,391	0,10214	0,8521	21,434	21,299	0,11558	1,0218	20,1597
	1000	5,75	216,12	0,1033	6,7204	209,3	206,78	0,0791	6,0776	200,626
	10000	58,81	2144,3	0,10253	713,28	1430,9	2050,2	0,0745	679,97	1370,13
	1E+05	590,9	21425	0,09984	20005	1420,4	20484	0,10522	19134	1350,35
	2E+05	1155	42847	0,10675	41393	1454,2	40965	0,10829	39581	1383,81
	5E+05	2883	107114	0,07334	105697	1417,1	102395	0,10445	101043	1351,7
	1E+06	5758	214203	0,10522	212787	1415,4	204803	0,11059	203447	1356,07
32	1	0,01	0,796	0,07411	0,3594	0,3606	0,791	0,07565	0,3706	0,34291
	100	1,025	29,82	0,10214	0,8655	28,85	28,226	0,19239	0,9506	27,1075
	1000	10,04	290,9	0,10637	7,0671	283,73	273,25	0,10253	5,9363	267,209
	10000	106,2	2889,2	0,52339	965,88	1922,8	2716,1	0,08141	921,45	1794,61
	1E+05	1003	28865	0,08026	26942	1923	27141	0,08141	25347	1794,45
	2E+05	2036	57728	0,0745	55771	1956,5	54285	0,08563	52431	1853,91
	5E+05	4499	144249	0,07488	142324	1924,3	135706	0,07757	133901	1805,75

CODIFICAR										
N	Nº Iter.	t_C (ms)	Memoria Global (ms)				Memoria Compartida (ms)			
			t_Tot	t_h_d	t_ker	t_d_h	t_Tot	t_h_d	t_ker	t_d_h
64	1	0,017	0,993	0,10675	0,3871	0,4969	0,917	0,07296	0,3813	0,4608
	20	0,352	0,1056	0,46118	0,0461	8,4396	8,3167	0,10562	0,465	7,74495
	40	0,815	17,337	0,07795	0,5445	16,712	16,155	0,10982	0,6797	15,3639
	200	3,552	85,287	0,08179	1,4489	83,754	78,846	0,10253	1,4769	72,2655
	500	11,73	211,86	0,10448	3,7179	208,03	194,05	0,08909	4,4172	189,54
	1000	19,98	419,71	0,07795	5,8387	413,79	389,3	0,08026	5,8109	380,4
	2000	36,42	844,54	0,1056	15,209	829,23	771,72	0,10829	11,624	759,982
	5000	95,68	2098,4	0,07526	31,645	2066,6	1926,4	0,08064	31,332	1894,99
	10000	198,5	4181,5	0,10214	1393,1	2783,3	3853	0,10829	1295,3	2557,57
	1E+05	1781	41836	0,07526	39058	2777,8	38507	0,12749	35946	2561,11
	2E+05	3545	83818	0,0768	80955	2863,6	77016	0,10675	74390	2625,91
	3E+05	5394	125643	0,10637	122860	2782,7	115519	0,10445	112955	2563,83
	5E+05	8993	209398	0,10176	206613	2785,2	192538	0,1116	189971	2567,11
	1E+06	18323	41921	0,08218	416466	2784,7	385060	0,1008	352509	2550,97
	2E+06	36718	837139	0,08909	834358	2781,6	770133	0,10483	767573	2559,59
	3E+06	46440	1E+07	0,10176	1E+06	2794,8	962975	0,10752	960118	2557,58
96	1	0,023	1,081	0,10406	0,379	0,596	1,0324	0,11098	0,4362	0,55411
	100	2,461	55,856	0,1033	0,8552	54,896	51,183	0,11136	0,8579	50,2122
	1000	23,7	550,35	0,08256	6,6747	543,59	501,69	0,10598	5,8295	495,753
	10000	264,7	5501,7	0,10675	1845,7	3655,9	5003,5	0,07565	1670,9	3332,48
	1E+05	2068	54890	0,09638	51247	3642,9	500015	0,10752	46686	3328,44
	2E+05	5215	109775	0,07872	106024	3750,5	1E+06	0,08141	96611	3421,21
	5E+05	13087	274424	0,07411	270771	3653	250056	0,07718	246723	3332,69
192	1	0,052	1,5053	0,1033	0,3959	1,0046	1,4366	0,10906	0,4155	0,91047
	100	4,471	96,574	0,10368	0,8801	95,589	86,937	0,11323	1,0779	85,7443
	1000	47,42	959,04	0,10291	5,8537	953,08	860,82	0,10253	5,8745	854,844
	10000	450,1	9566,1	0,10445	3194,4	6371,6	8594,3	0,10711	2878,2	5716,05
	1E+05	4350	95622	0,10714	89265	6356,7	85932	0,1175	80214	5717,31
	2E+05	9142	191261	0,11328	184718	6543,5	171854	0,08333	165971	5882,8
288	1	0,068	1,9503	0,08332	0,4093	1,4554	1,7733	0,07872	0,3855	1,30753
	100	6,954	138,43	0,07642	0,8652	137,49	0,0952	0,09523	0,8886	122,932
	1000	70,47	1378,4	0,10982	6,0745	1372,2	1232,6	0,10501	6,0488	1226,42
	10000	787	13764	0,10483	4600	9164,2	12318	0,08909	4120,7	8196,82
	1E+05	6777	137651	0,07488	128482	9169,5	123101	0,11174	114904	8196,23
384	1	0,089	2,3731	0,10483	0,0399	1,867	2,1439	0,08755	0,3852	1,66964
	100	9,316	182,32	0,10061	0,8863	181,33	163,08	0,06989	0,8517	162,16
	1000	95,98	1816,3	0,07641	5,7835	1810,4	1622,4	0,14928	5,9201	1616,36
	10000	984,8	18152	0,1102	6050,9	12101	16220	0,10522	5412,7	10806,8
	1E+05	9461	181429	0,10176	199349	12080	162116	0,18279	151316	10779,1

CODIFICAR										
N	Nº Iter.	t_C (ms)	Memoria Global (ms)				Memoria Compartida (ms)			
			t_Tot	t_h_d	t_ker	t_d_h	t_Tot	t_h_d	t_ker	t_d_h
512	1	0,158	2,9933	0,08678	0,3875	2,5171	2,713	0,06144	0,3686	0,36865
	2	0,281	5,4924	0,10099	0,4159	4,974	4,9575	0,09716	0,4063	4,45211
	5	0,607	12,931	0,07565	0,4155	12,437	11,631	0,10022	0,4278	11,0999
	10	1,293	25,26	0,07066	0,4362	24,751	22,581	0,09984	0,4408	22,039
	20	2,47	49,901	0,0745	0,4723	49,353	44,765	0,10253	0,5464	44,1136
	50	6,083	124,21	0,07488	0,6086	123,53	110,56	0,0745	0,5994	109,884
	100	12,09	248,28	0,09907	0,8809	247,29	221,04	0,07066	0,8721	220,093
	1000	130,7	2474	0,07565	5,9597	2468	2201,9	0,22656	5,9601	2195,71
	10000	1308	24715	0,07027	8235,6	16480	22001	0,07219	7341,7	14659,6
	1E+05	12309	247164	0,10138	230699	16465	220032	0,09869	205370	14662,2
	2E+05	24383	494538	0,1248	477599	16939	440073	0,07065	424982	15091
640	1	0,155	3,7798	0,10214	0,4166	3,259	3,4426	0,1033	0,4105	2,92686
	2	0,31	7,1236	0,13172	0,5172	6,4731	6,3191	0,10061	0,4097	5,80725
	5	0,766	16,768	0,2132	0,4547	16,098	15,045	0,10675	0,4927	14,4438
	10	1,52	32,837	0,19046	0,457	32,188	29,457	0,08179	0,4762	28,8976
	20	3,014	64,775	0,0745	0,4831	64,216	58,169	0,09907	0,49	57,5786
	50	7,67	161,09	0,08717	0,6724	160,33	144,39	0,07219	0,6067	143,71
	100	21,02	321,83	0,1056	0,9112	320,81	288,52	0,10099	0,8863	287,528
	1000	154,4	3211,2	0,07795	6,0826	3205	2876,4	0,07027	5,7424	2870,54
	10000	1617	32074	0,07334	10697	21377	28765	0,09984	9598,9	19165,6
	1E+05	15089	320854	0,10448	3E+06	21369	287572	0,10022	268421	19151,4
	2E+05	30324	640121	0,12864	618183	21937	575156	0,10176	555454	19702,1
768	1	0,183	5,204	0,11597	0,4124	4,6737	4,1775	0,07334	0,3994	3,70331
	2	0,366	8,6151	0,10364	0,4366	8,0705	7,7772	0,07258	0,4055	7,29755
	5	0,91	20,649	0,0768	0,4349	20,136	18,662	0,07296	0,4182	18,1694
	10	1,814	40,728	0,10406	0,4727	40,149	36,804	0,0768	0,4451	36,28
	20	3,708	80,874	0,10445	0,5134	80,255	73,127	0,10522	0,4892	72,5309
	50	9,077	201,15	0,1079	0,7258	200,31	181,76	0,10176	0,6367	181,016
	100	24,02	401,68	0,10867	0,9243	400,65	363,1	0,11021	0,8855	362,103
	1000	188,2	4011,8	0,11866	6,5038	4005,1	3623,1	0,07603	5,8026	3617,26
	10000	1898	40094	0,10483	13364	26730	36222	0,11674	12083	24138,1
	20000	3687	80169	0,07987	53502	25557	72447	0,09984	48340	24107,5
	30000	5510	120220	0,07642	93558	26662	108688	0,07258	84591	24107,3

CODIFICAR										
N	Nº Iter.	t_C (ms)	Memoria Global (ms)				Memoria Compartida (ms)			
			t_Tot	t_h_d	t_ker	t_d_h	t_Tot	t_h_d	t_ker	t_d_h
896	1	0,214	5,649	0,10368	0,4282	5,1153	5,1491	0,08793	0,4136	4,64603
	2	0,426	10,671	0,10368	0,4516	10,115	9,7179	0,07258	0,04	9,24367
	5	1,255	25,742	0,07526	0,4289	25,236	23,566	0,10061	0,432	23,0312
	10	2,117	50,832	0,0768	0,4462	50,307	46,547	0,10061	0,455	45,9891
	20	4,283	101,27	0,07718	0,4908	100,71	92,587	0,07334	0,4792	92,0324
	50	10,72	252,01	0,10445	0,6417	251,27	230,32	0,07411	0,6182	229,622
	100	21,53	505,54	0,07296	2,8032	502,66	460,33	0,07526	0,8663	459,382
	1000	224	5028,6	0,07718	5,9493	5022,6	4595,1	0,0745	5,9194	4589,07
	10000	2217	50293	0,07949	16775	33518	45941	0,09946	15315	30626,5
	20000	4485	100551	0,11789	67093	33457	91890	0,07142	61319	30570,8
	30000	6414	150840	0,10598	117377	33463	137838	0,07258	107259	30579,7
1024	1	0,255	6,7799	0,0768	0,4136	6,288	6,3352	0,10406	0,4305	5,7988
	5	1,237	31,709	0,07872	0,4324	31,196	29,3	0,10368	0,4512	28,744
	10	2,438	62,974	0,10406	0,4719	62,396	58,055	0,10522	0,4846	57,4634
	50	12,2	312,06	0,10445	0,6467	311,31	287,57	0,10445	0,6432	286,82
	100	24,3	623,87	0,10214	0,9043	622,86	574,83	0,10368	0,8993	573,828
	1000	261,5	6228,5	0,07987	5,8291	6222,6	5740,5	0,12019	5,8069	5734,52
	10000	2526	62301	0,10368	20769	41532	57400	0,10291	19139	38261
	20000	4994	124610	0,07757	83151	41459	114762	0,10138	76570	38191,4
	30000	7324	186918	0,07757	145463	41455	172153	0,10061	133970	38183,2

Si se observan los diferentes datos, se puede comprobar que el tiempo en C siempre es mucho menor que en CUDA. Además se comprueba que el tiempo de trasferencia entre el host y el device siempre es muy pequeño y más o menos se mantiene constante.

Por el contrario el tiempo total de ejecución en CUDA y el del kernel aumentan, pero el total lo hace siempre en una proporción similar, el del kernel sufre grandes variaciones. Finalmente la instrucción que consume mayor tiempo es la transferencia final del device al host, sobre todo para valores de iteraciones pequeños, ya que para valores grandes se mantiene constante y el factor limitante es el kernel.

Para entender mejor esta explicación se procede a realizar un nuevo gráfico, de manera aleatoria se precede a realizar el ensayo con el texto plano de 288 bytes y Memoria Global, pero en términos generales todos los resultados siguen una tendencia similar.

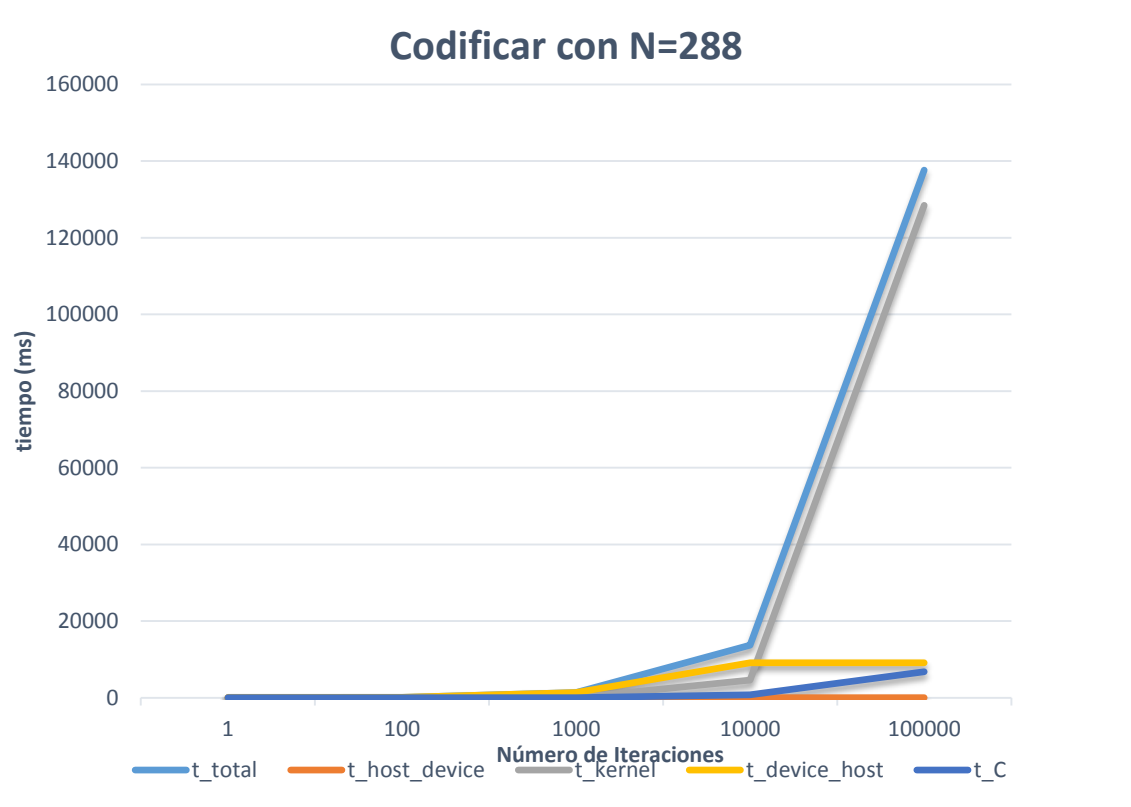

Figura 7.5 Desglose tiempos de ejecución para N=288

Como se demuestra en la gráfica el tiempo de transferencia entre el device y el host es una contribución muy importante hasta a partir de un cierto número de iteración en el cual el tiempo del kernel se convierte en predominante.

Pero también es cierto que en todos ensayos que se muestran en la tabla anterior, siempre aparece algún intervalo en el que el tiempo del kernel es menor que el tiempo de C. De esta manera, el programa consigue acelerarse, pero el problema son los traspasos de memoria del device (GPU) a la (CPU) que lastran el proceso.

Este hecho se puede comprobar, por ejemplo representando únicamente los tres primeros puntos de la gráfica anterior en la figura 7.6. Mediante esta ampliación se observa como en ese rango de valores el tiempo de C es superior al del kernel en CUDA, de esta manera se comprueba también que mejora cuanto más grande sea la información a cifrar.

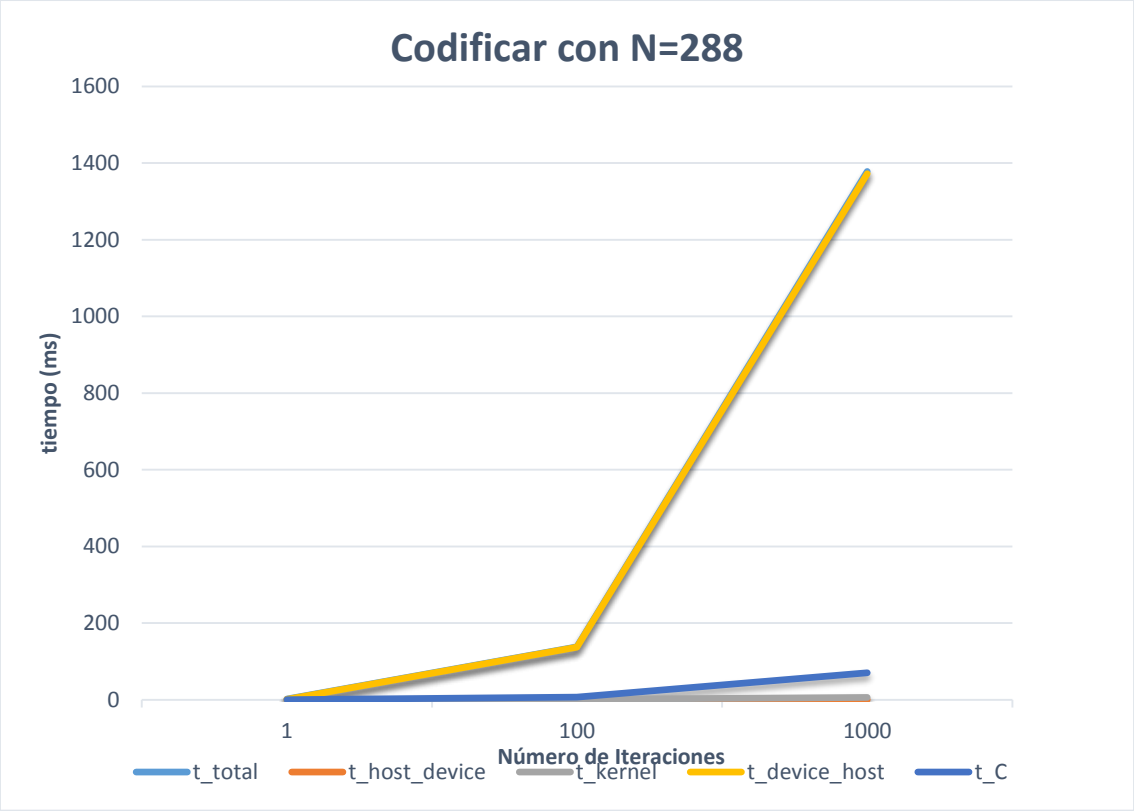

Figura 7.6 Ampliación primeros tiempos de ejecución para N=288

Este análisis se puede extrapolar para distintos tamaños de texto plano y uso de memoria, ya que como se puede comprobar este punto aparece en todos los tamaños de texto en algún intervalo. Además este intervalo nunca aparece para valores ni muy pequeños ni muy altos de iteraciones.

Capítulo 8. Conclusiones

8.1. Conclusiones

En primer lugar, se debe decir que se han conseguido cumplir todos los objetivos que se indicaban en el primer capítulo. Ya que se ha logrado implementar el algoritmo de cifrado AES tanto en C como en CUDA en el modo ECB para cifrar y descifrar, a pesar de todas las complicaciones de las que se partía inicialmente.

A continuación, se exponen todas las conclusiones que se pueden extraer tras analizar ampliamente los resultados obtenidos en el capítulo anterior. Así como la justificación de alguna de las anomalías que han aparecido en los datos.

Para justificar los datos obtenidos se debe hacer referencia al algoritmo utilizado, en este caso se trata de AES. El principal inconveniente de AES es que utiliza un procedimiento recursivo, lo que dificulta su paralelización. Sin embargo se decide implementar este algoritmo debido a la importancia de los métodos de cifrado en el mundo actual y a su relación con la potencia y velocidad de cálculo. Además en el caso de este TFG se implementa la versión estándar reconocida por el NIST en el FIPS correspondiente, mientras que otras investigaciones que han obtenido buenos resultados han implementado AES mediante la matriz T-BOX comentada anteriormente. Se decidió así, porque precisamente se quería estudiar el comportamiento de la versión estándar del algoritmo.

Otro factor a tener en cuenta es la cpacidad de la GPU y el ordenador utilizado. Uno de los objetivos del TFG era analizar el rendimiento en GPUs de bajo coste, sin embargo, si se utiliza una GPU más potente, o incluso un coprocesador Tesla, posiblemente los resultados hubieran sido distintos. Este análisis se propone como trabajo futuro.

Con todas estas premisas se deben analizar los resultados. Gracias a estos resultados se pueden extraer una serie de conclusiones muy valiosas, que incluso se pueden extrapolar a cualquier otro programa en CUDA.

De modo general y a la vista de los resultados obtenidos se puede decir que el procedimiento utilizado en la implementación mejora al aumentar el tamaño de texto plano. Este hecho concuerda con que cuanto más grande sea el tamaño de la información transferida mejores datos se obtienen en cuanto a velocidad de procesamiento y ratio.

Aunque también es cierto que a partir de un cierto tamaño de texto plano, unos 640 bytes aproximadamente, los resultados comienzan a empeorar o por lo menos a estancarse. Este suceso se debería haber estudiado en mayor profundidad, pero debido a una limitación de la GPU los ensayos no se pueden realizar para tamaños de texto mayores que 1024 bytes. Así pues, este estudio se podría ampliar en un trabajo futuro.

Este comportamiento en los resultados se puede deber a varias causas. La primera es que exista algún tipo de limitación en el tamaño de la trasferencia de los datos. Otra posible causa es el modo en el que se han realizado las diferentes implementaciones, porque solo se puede realizar hasta 1024 bytes, y es posible que al acercarse al límite de núcleos de la GPU el método se vuelva más impreciso debido a que el procedimiento comience a saturar.

Del mismo modo se debe comentar la diferencia que existe entre los resultados obtenidos en el ratio para cifrar y descifrar. Se recuerda que el ratio calculado muestra la relación que existe entre los tiempos de ejecución en CUDA y C, de esta manera cuanto más grande sea el ratio peor es el resultado y por lo tanto mayor es el tiempo en CUDA respecto el de C.

Así pues, se puede ver como aproximadamente el valor más alto del ratio en el caso de la codificación es 35, mientras que para la descodificación ese mismo ratio toma un valor aproximado de 10, por lo que el programa en el modo descodificando es mucho más eficiente. Además este hecho concuerda, porque el programa para descodificar tarda más en ejecutarse tanto en C como en CUDA, así pues este hecho transmite la idea de que cuanto más costoso computacionalmente sea el programa y más tarde en ejecutarse tanto en C como en CUDA, la paralelización tendrá un rendimiento mayor y será más eficiente.

Al mismo tiempo en este TFG también se ha estudiado el uso de la Memoria Global y la Compartida a la hora de cifrar y descifrar. Tras analizar los resultados se comprueba que el uso de la Memoria Compartida es más eficiente, en concreto en la implementación para codificar se consigue una mejora del 8,62% en el ratio y en el caso de la descodificación una mejora del 2,04% también en el ratio. Esto coincide con la definición que se dio en su momento de Memoria Compartida, ya que por definición la Memoria Compartida siempre es más rápida. A la vista de estos resultados también se puede concluir que la mejora en el caso del cifrado es mayor, porque los datos iniciales eran mucho peores y por lo tanto presentan un mayor margen de mejora.

De la misma manera se ha comprobado mediante representación gráfica (figura 7.2 y 7.4) que la velocidad de procesamiento es mayor mediante el uso de la Memoria Compartida que con la Memoria Global, aunque también se debe decir que ambas velocidades son mucho menores que la obtenida en el caso de C.

Finalmente, si se atiende al último ensayo realizado, se puede ver el consumo de tiempo en cada transferencia de memoria y en la ejecución del kernel. De ello se puede decir como conclusión que el motivo por cual el programa tarda tanto en ejecutarse es la última transferencia de memoria de la GPU a la CPU, aunque también es cierto que a partir de un cierto número de iteraciones el valor se estanca, pero para números pequeños de iteraciones supone un auténtico lastre.

Si se hace referencia al kernel, se puede decir que para valores pequeños de iteraciones su tiempo de ejecución es pequeño pudiendo ser incluso menor que el de C. Pero desde el momento en el que se estanca el tiempo de la transferencia de memoria, el tiempo de ejecución del kernel se dispara. De la misma manera, en cada uno de estos ensayos siempre aparece un intervalo de iteraciones en el que el tiempo de ejecución del kernel es menor que el tiempo de ejecución de C, pero finalmente debido a las trasferencias de memoria el resultado no es viable.

Así pues, aunque en este caso no se ha conseguido acelerar lo suficiente el programa en CUDA los datos obtenidos son coherentes con la técnica utilizada. Por lo que se puede suponer que en un programa de mayor longitud y más paralelizable se obtendrá una notable reducción de tiempo de cómputo. Pero como ya se ha comentado, el objetivo del presente TFG es estudiar en CUDA el comportamiento del AES en su versión estándar reconocida por el NIST.

Además se debe recordar que este análisis se realiza sin tener en cuenta el cálculo de las claves de ronda, ya que este proceso no se podía paralelizar y se realiza mediante la CPU tanto en C como en CUDA, así pues ese tiempo de ejecución se ha excluido de todas las mediciones temporales.

Resumiendo, si se quiere obtener unos buenos resultados en CUDA, se debe partir de un código o algoritmo extenso que se pueda paralelizar fácilmente. También se debe prestar

especial atención a la transferencia de los datos, sobre todo de la GPU a la CPU, porque en el sentido inverso (CPU a GPU) no consume mucho tiempo, se debe intentar buscar un cierto equilibrio en los resultados. Finalmente el último factor a tener en cuenta a la hora de programar es que siempre es preferible utilizar la Memoria Compartida en vez de la Global. Así pues, estas conclusiones se pueden extender a cualquier programa en CUDA, por lo que este TFG puede servir como base para diferentes implementaciones.

Para finalizar, destacar que, aunque en este caso específico y muy particular, no se haya conseguido una implementación totalmente óptima, si se puede considerar el uso de la GPU como una solución viable, rápida y especialmente económica para acelerar algoritmos. Del mismo modo también se puede considerar la programación en CUDA y el uso de la tecnología Nvidia una buena opción para introducir en el mundo de la programación paralela a cualquier programador.

Capítulo 9. Trabajos futuros

9.1. Trabajos futuros

En este capítulo se desarrollan brevemente los trabajos futuros que se pueden desarrollar teniendo el presente TFG como base, ya que se puede utilizar para profundizar más en las técnicas de programación paralela y por lo tanto en CUDA. Algunos de estos trabajos se plantean a continuación:

- Implementar la otra versión del AES que se ha comentado anteriormente en este TFG, es decir, la que utiliza la T-BOX y mediante la que otros estudios han obtenido buenos resultados con CUDA.
- Buscar un modo de intentar reducir el tiempo de trasferencia entre la GPU y la CPU, por ejemplo intentar implementar una solución que permita transmitir información entre la GPU y la CPU mientras que se está codificando o descodificando.
- Realizar más ensayos con un tamaño de texto plano cercano a los 640 bytes, por lo que se tendrán más datos del ratio y la velocidad de procesamiento. De este modo se pretende conocer si la anomalía comentada anteriormente está fundamentada o por el contrario se trata de un dato atípico.
- Probar el programa implementado en una GPU distinta y por lo tanto con otras características, para poder obtener así otra serie datos y compararlos con los analizados anteriormente en este TFG. Además se podría relacionar así la potencia de la tarjeta con los resultados obtenidos en cada ensayo.
- Desarrollar el programa con una configuración distinta de bloque e hilo, para que pueda admitir así tamaños de texto plano mayores a 1024 bytes. De este modo, se podrían representar las gráficas del ratio y la velocidad de procesamiento con un mayor número de puntos.
- Ejecutar diferentes programas con distintas configuraciones de bloques e hilos, para relacionar los diferentes resultados obtenidos en función del número de bloques y threads.
- Realizar este mismo estudio con otros tamaños de clave, para conocer la evolución de los resultados en CUDA para diferentes tamaños y por lo tanto distinto número de rondas.
- Implementar el algoritmo AES en el sistema empotrado de Nvidia Jetson TK1 [54], que dispone de una pequeña GPU con 192 núcleos.
- Intentar proteger el código tanto en lenguaje C como en CUDA ante ataques de canal lateral.
- Implementar AES mediante programación paralela en otra GPU, pero que no utilice CUDA, por ejemplo la tecnología equivalente de AMD o un estándar abierto.

Capítulo 10. Presupuesto

10.1. Planificación

Para realizar de un modo satisfactorio este TFG es necesario planificar las tareas que se realizan. Así como analizar de manera aproximada el tiempo que se invierte en cada una de esas tareas. Realizar una planificación es algo esencial en la ejecución de cualquier proyecto complejo. Así pues a continuación se enumeran y explican cada una de las fases en las que se ha dividido este proyecto:

- **Fase 1. Búsqueda genérica de información:**
  Una vez definido el campo de trabajo se procede a buscar información genérica sobre él, es decir, se empezó a investigar sobre el mundo de la criptografía y en particular sobre el algoritmo de cifrado AES. De la misma manera también se buscó información sobre Nvidia y CUDA. A su vez también se buscaron trabajos y proyectos previos que estuviesen relacionados con este tema de estudio.
  Tiempo estimado: 25 horas.
- **Fase 2. Aprendizaje del lenguaje C:**
  Durante el desarrollo de esta fase se procedió a aprender el lenguaje C. En un principio se comenzó por implementar ejercicios simples en este lenguaje, para ello se usaron principalmente tutoriales y apuntes de asignaturas. Finalmente, pero con la misma filosofía se profundizó más en el tema, por ejemplo con el uso de funciones. En esta fase también se aprendió el manejo de Qt creator y Visual Studio como herramientas de desarrollo.
  Tiempo estimado: 60 horas.
- **Fase 3.Desarrollo de las implementaciones en C:**
  Depués de asimilar los conceptos básicos del lenguaje C, se procedió a comenzar a desarrollar las implementaciones del algoritmo AES en CUDA, para lo cual se realizaron un gran número de pruebas hasta llegar a su resolución final. Del mismo modo, también se realizaron algunas optimizaciones y se utilizaron funciones en el desarrollo del programa.
  Tiempo estimado: 110 horas
- **Fase 4. Aprendizaje de CUDA:**
  A lo largo de esta fase se adquirieron conocimientos en el lenguaje CUDA, de igual modo se comprendió el funcionamiento de la tecnología de Nvidia para acelerar y paralelizar algoritmos. Para ello se utilizaron tutoriales y ejercicios básicos realizados mediante CUDA.
  Tiempo estimado: 60 horas.
- **Fase 5. Desarrollo de las implementaciones en CUDA:**
  Tras conocer los fundamentos de CUDA, se comenzó a desarrollar las diferentes implementaciones del algoritmo. Hasta llegar a la implementación final, se ejecutaron diferentes implementaciones para solucionar los problemas que iban surgiendo.
  Tiempo estimado: 120 horas.
- **Fase 6. Toma de datos:**
  En esta etapa se tomaron todos los datos a través de los dos procedimientos explicados y para las diferentes implementaciones utilizadas. Gracias a esos datos se han obtenido las conclusiones que se han detallado anteriormente.
  Tiempo estimado: 45 horas.
- **Fase 7. Desarrollo de la memoria:**
  Esta etapa es la más larga y se ha desarrollado paralelamente a todas las implementaciones realizadas en este TFG y a la toma de datos. Esta fase ha sido la más

extensa y tediosa, debido a que en la memoria aparece tanto una parte teórica como práctica.
Tiempo estimado: 320 horas.

- **Fase 8. Revisión final:**
  En esta etapa final se revisó toda la memoria y se ejecutó su maquetación final
  Tiempo estimado: 10 horas

Se debe decir que todos estos tiempos son estimados y que tampoco se ha invertido ese tiempo de forma continua en cada etapa, ya que en muchas ocasiones se han ido intercalando diferentes etapas. En la siguiente tabla se muestran los periodos de inicio y fin de cada etapa así como su duración.

Tabla 10.1 Periodos del proyecto

Etapa	Fecha de inicio	Fecha de fin	Duración
Fase 1	01/04/2014	07/04/2014	25
Fase 2	07/04/2014	28/04/2014	60
Fase 3	28/04/2014	01/06/2014	110
Fase 4	23/06/2014	15/07/2014	60
Fase 5	15/07/2014	05/08/2014	120
Fase 6	01/09/2014	16/09/2014	45
Fase 7	01/06/2014	18/09/2014	320
Fase 8	18/09/2014	19/09/2014	10

Si sumamos todas las horas dedicadas, se han utilizado en la ejecución de este proyecto un total de **750 horas**. Esta planificación se puede observar en el siguiente diagrama de Gantt:

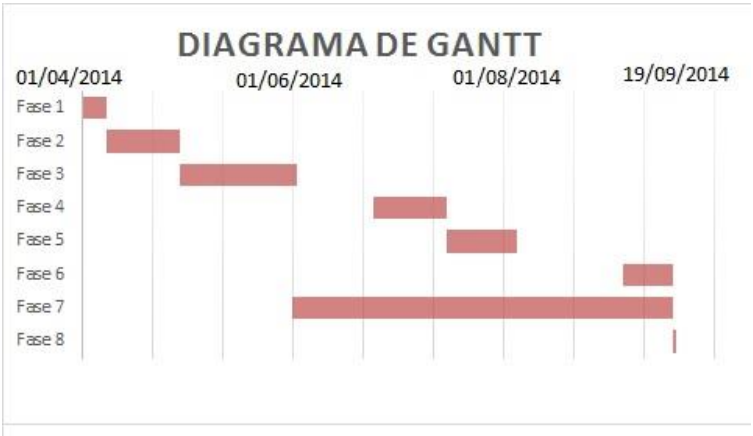

Figura 10.1 Diagrama de Gantt

Como se puede ver las tareas críticas son la búsqueda de la información, el aprendizaje e implementación en C, la realización de la memoria y finalmente la revisión final del trabajo.

10.2. Presupuesto

A continuación se desarrolla un presupuesto económico detallado, para conocer el coste que ha implicado la realización de este TFG. De esta manera se incluyen tanto los costes del personal como los correspondientes a amortización de equipos y licencias.

10.2.1. Costes de personal

En la siguiente tabla se especifica el tiempo de dedicado por la autora y el tutor, así como su coste:

Tabla 10.2 Honorarios del personal

Costes de personal				
Personal	Cargo	Dedicación (h)	Coste (€/h)	C_{pers} (€)
Tutor	Ingeniero	50	40	2000
Autora	Estudiante	750	15	11250
			Total	13250

10.2.2. Amortización de equipos y licencias

En este apartado se comentan el coste y la amortización de los equipos y licencias utilizados. Para calcular la amortización de los equipos se utiliza la siguiente fórmula:

$$C_{amort} = \frac{A}{B} \cdot C \cdot D$$

Donde A representa el número de tiempo en el que el equipo se ha utilizado, B es el periodo de depreciación, C indica el coste del equipo y finalmente D es el porcentaje de utilización, normalmente toma el valor del 100 %.

Tabla 10.3 Costes de amortización

Costes de amortización					
Equipo	A (meses)	B(meses)	C(€)	D (%)	C_{amort}(€)
Ordenador laboratorio	2	12	600	100	100
Ordenador personal	6	12	500	100	250
Nvidia GeForce GTX 550 Ti	2	12	140	100	23.33
Licencias de software	A (meses)	B(meses)	C(€)	D (%)	C_{amort}(€)
Microsoft Office	6	12	150	100	75
Visual Studio 2010	4	-	-	100	0
Visual Studio 2012	4	-	-	100	0
Qt creator	2	-	-	100	0
				Total	448.33

Se debe comentar que las dos licencias de Visual Studio no presentan ningún coste porque se obtienen a través del acuerdo de la universidad con Microsoft. Aademás Qt creator tampoco tiene coste porque es libre.

10.2.3. Coste total

Así pues el coste total del proyecto asciende a:

$$\boldsymbol{C_{total}} = C_{pers} + C_{amort} = 13250 + 448.33 = \mathbf{13698.33}\ €$$

Para concluir se debe decir que en esta ocasión se han despreciado tanto los costes de oficina como los indirectos.

ANEXOS

ANEXO A: Cifrado en AES con C

```
// Programa que codifica los cuatro vectores de TEST del NIST
// Invoco las librerías
#include <stdio.h>
#include <time.h>
#define N 64
// Defino las variables que se usan tanto en el main como en las funciones
int m;
unsigned int vector_claves [176];
unsigned int vector_estado[N]={
0x6b,0x2e,0xe9,0x73,0xc1,0x40,0x3d,0x93,0xbe,0x9f,0x7e,0x17,0xe2,0x96,0x11,0x2a,  //Vector 1
0xae,0x1e,0x9e,0x45,0x2d,0x03,0xb7,0xaf,0x8a,0xac,0x6f,0x8e,0x57,0x9c,0xac,0x51,  //Vector 2
0x30,0xa3,0xe5,0x1a,0xc8,0x5c,0xfb,0x0a,0x1c,0xe4,0xc1,0x52,0x46,0x11,0x19,0xef,  //Vector 3
0xf6,0xdf,0xad,0xe6,0x9f,0x4f,0x2b,0x6c,0x24,0x9b,0x41,0x37,0x45,0x17,0x7b,0x10}; //Vector 4
//Tabla de sustitucion de bytes (S-box)
int sbox[256] =   {
    //0     1    2      3     4    5     6     7      8    9     A      B    C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};//F
//Defino las funciones que voy a programar
void addRoundKey();
void subBytes();
void shiftRows();
void mixColumns();
//Empieza el cuerpo principal del programa (main)
int main ()
{
//Defino las variables que solo se utilizan en el main
//Defino las variables que utilizo para calcular el tiempo
clock_t begin, end;
double time_spent;
//Defino los contadores
       int i,j,n,l, temporal;
       unsigned int clave[4][4]={
              {0x2b,0x28,0xab,0x09},
              {0x7e,0xae,0xf7,0xcf},
              {0x15,0xd2,0x15,0x4f},
              {0x16,0xa6,0x88,0x3c}
               };
       unsigned int matriz_claves[4][44];
//En el vector de claves se almacenan todas las claves, las 16 primeras variables
son la primera clave de cifrado, las siguientes 16 la segunda y así sucesivamente
        unsigned int sub_claves[4][8];
//Rcon es la matriz con las potencias de 2 para el cálculo de las subclaves
int Rcon [4][11]={
                {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36},
                {0X00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
               };
       // A continuación calculo todas las subclaves en la CPU
```

```
                for(i=0;i<=3;i++)
                    {
                        for(j=0;j<=3;j++)
                        {
                           matriz_claves[i][j]=clave[i][j];
                        }
                    }
                    for (n=0;n<=9;n++)
                    {
                    for(i=0;i<=3;i++)
                    {
                        for(j=0;j<=3;j++)
                        {
                            sub_claves[i][j]=clave[i][j];
                        }
                    }
                    for (i=0;i<=3;i++)
                    {
                        sub_claves[i][4]=clave[i][3];
                    }
                    temporal=sub_claves[0][4];
                    sub_claves[0][4]=sub_claves[1][4];
                    sub_claves[1][4]=sub_claves[2][4];
                    sub_claves[2][4]=sub_claves[3][4];
                    sub_claves[3][4]=temporal;
                    for(i=0;i<=3;i++)
                    {
                        sub_claves [i][4]=sbox[sub_claves[i][4]];
                    }
                   sub_claves[0][4]=sub_claves[0][0]^sub_claves[0][4]^ Rcon[0][n];
                    sub_claves[1][4]=sub_claves[1][0]^sub_claves[1][4]^ (0x00);
                    sub_claves[2][4]=sub_claves[2][0]^sub_claves[2][4]^ (0x00);
                    sub_claves[3][4]=sub_claves[3][0]^sub_claves[3][4]^ (0x00);

                    for(i=0;i<=3;i++)
                    {
                        sub_claves[i] [5]=sub_claves[i] [1] ^sub_claves[i] [4];
                        sub_claves[i] [6]=sub_claves[i] [2] ^sub_claves[i] [5];
                        sub_claves[i] [7]=sub_claves[i] [3] ^sub_claves[i] [6];
                    }
                    for(i=0;i<=3;i++)
                    {
                        clave[i][0]=sub_claves[i][4];
                        clave[i][1]=sub_claves[i][5];
                        clave[i][2]=sub_claves[i][6];
                        clave[i][3]=sub_claves[i][7];
                    }
                    for(i=0;i<=3;i++)
                    {
                        matriz_claves[i][4*(n+1)]=clave[i][0];
                        matriz_claves[i][4*(n+1)+1]=clave[i][1];
                        matriz_claves[i][4*(n+1)+2]=clave[i][2];
                        matriz_claves[i][4*(n+1)+3]=clave[i][3];
                    }
                    }
//Meto todas las claves en un vector para facilitar las operaciones durante la codificación
                     for(j=0;j<=10;j++)
                        {
                         for(i=0;i<=3;i++)
                          {
                        vector_claves [16*j+i]=matriz_claves[0][4*j+i];
                        vector_claves[16*j+4+i]=matriz_claves[1][4*j+i];
                        vector_claves[16*j+8+i]=matriz_claves[2][4*j+i];
```

```
                        vector_claves[16*j+12+i]=matriz_claves[3][4*j+i];
                          }
                        }
//Inicio el contador de tiempo
begin = clock();
//Inicio el proceso de cifrado
                                m=0;
                                addRoundKey ();
                                for (l=0;l<=8;l++)
                                {
                                        subBytes ();
                                        shiftRows();
                                        mixColumns();
                                        m=m+1;
                                        addRoundKey();
                                }
                                        subBytes();
                                        shiftRows();
                                        m=m+1;
                                        addRoundKey();

//Muestro el tiempo de ejecución
  end = clock();
  time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  printf("El tiempo de ejecucion es %lf \n",time_spent);
//Muestro los vectores codificados
  printf("Los vectores de Test del NIST cifrados son:\n");
                  for (i=0;i<16;i++)
                  {
                      printf("%x   ",vector_estado[i]);
                  }
                          printf("\n");
                          for (i=16;i<32;i++)
                  {
                      printf("%x   ",vector_estado[i]);
                  }
                          printf("\n");
                          for (i=32;i<48;i++)
                  {
                      printf("%x   ",vector_estado[i]);
                  }
                          printf("\n");
                          for (i=48;i<64;i++)
                  {
                      printf("%x   ",vector_estado[i]);
                  }
                          printf("\n");
                          system ("pause");
        return 0;
}
//Programo las funciones utilizadas
void addRoundKey ()
{ int i;
//Calculo la operación XOR entre el estado y la clave correspondiente
        for (i=0; i<16; i++)
        {
                vector_estado[i]=vector_estado[i]^vector_claves[(16*m)+i];
                vector_estado[i+16]=vector_estado[i+16]^vector_claves[(16*m)+i];
                vector_estado[i+32]=vector_estado[i+32]^vector_claves[(16*m)+i];
                vector_estado[i+48]=vector_estado[i+48]^vector_claves[(16*m)+i];
        }
}
```

```
void subBytes()
{
	int i;
// Sustituyo cada elemento del estado por su equivalente en la matriz de sustitución de bytes
	for (i=0;i<N;i++)
	{
		vector_estado[i]=sbox[vector_estado[i]];
	}
}
void shiftRows()
{
	int i;
	unsigned int estado_auxiliar [N];
	//Realizo las rotaciones circulares en las filas
	for(i=0;i<4;i++)
{
	//Primera fila
	estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
	estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
	estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
	estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
	//Segunda fila
	estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+5];
	estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+6];
	estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+7];
	estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+4];
	//Tercera fila
	estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
	estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
	estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
	estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
	//Cuarta fila
	estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+15];
	estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+12];
	estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+13];
	estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+14];
}

	for (i=0;i<N;i++)
	{
		vector_estado [i]=estado_auxiliar[i];
	}
}
void mixColumns ()
{
	unsigned char Tmep,Tme,time;
// Multiplico cada columna por una matriz concreta en el Campo de Galois
// xtime es un macro que devuelve el producto con modulo {1b} de {02} y el byte argumento
    #define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

	int i,j;
	for (i=0;i<4;i++)
	{
	   for (j=0;j<4;j++)
			{

time=vector_estado[(16*i)+j];
Tmep = vector_estado[(16*i)+j] ^ vector_estado[(16*i+4)+j] ^
vector_estado[(16*i+8)+j] ^ vector_estado[(16*i+12)+j] ;
Tme = vector_estado[(16*i)+j] ^ vector_estado[(16*i+4)+j] ;
Tme = xtime(Tme);
vector_estado[(16*i)+j] ^= Tme ^ Tmep ;
Tme = vector_estado[(16*i+4)+j] ^ vector_estado[(16*i+8)+j] ;
```

```
Tme = xtime(Tme);
vector_estado[(16*i+4)+j] ^= Tme ^ Tmep ;
Tme = vector_estado[(16*i+8)+j] ^ vector_estado[(16*i+12)+j] ;
Tme = xtime(Tme);
vector_estado[(16*i+8)+j] ^= Tme ^ Tmep ;
Tme = vector_estado[(16*i+12)+j] ^ time ; Tme = xtime(Tme);
vector_estado[(16*i+12)+j] ^= Tme ^ Tmep ;
                                    }
      }
}
```

ANEXO B: Descifrado en AES con C

```
// Programa que descodifica los cuatro vectores de TEST del NIST
// Invoco las librerías
#include <stdio.h>
#include <time.h>
#define N 64
// Defino las variables que se usan tanto en el main como en las funciones
int m;
unsigned int vector_claves [176];
unsigned int vector_estado[N]={
0x3a,0x0d,0xa8,0x24,0xd7,0x7a,0x9e,0x66,0x7b,0x36,0xca,0xef,0xb4,0x60,0xf3,0x97,  //Vector 1
0xf5,0x03,0xe7,0x96,0xd3,0xb9,0x85,0xfd,0xd5,0x69,0x89,0xba,0x85,0x9d,0x5a,0xaf,  //Vector 2
0x43,0x59,0x88,0xed,0xb1,0x8e,0x1b,0x03,0xcd,0xce,0x00,0x06,0x7f,0x23,0xe3,0x88,  //Vector 3
0x7b,0x27,0x82,0x04,0x0c,0xe8,0x23,0x72,0x78,0xad,0x20,0x5d,0x5e,0x3f,0x71,0xd4}; //Vector 4
//Defino las funciones que voy a programar
void addRoundKey();
void inv_shiftRows();
void inv_subBytes();
void inv_mixColumns();
//Empieza el cuerpo principal del programa (main)
main()
{//Defino las variables que solo se utilizan en el main
             //Defino las variables que utilizo para calcular el tiempo
             clock_t begin, end;
             double time_spent;
//Defino los contadores
  int i, j, n, l, temporal;
//Tabla de sustitucion de bytes (S-box)
int sbox[256] =   {
    //0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};//F

unsigned int clave[4][4]={
     {0x2b,0x28,0xab,0x09},
     {0x7e,0xae,0xf7,0xcf},
     {0x15,0xd2,0x15,0x4f},
     {0x16,0xa6,0x88,0x3c}
        };
unsigned int matriz_claves[4][44];
```

```
//En el vector de claves se almacenan todas las claves,las 16 primeras variables son la última clave de descifrado, las siguientes 16 la penúltima clave y así sucesivamente
        unsigned int sub_claves[4][8];
//Rcon es la matriz con las potencias de 2 para el cálculo de las subclaves
        int Rcon [4][11]={
                {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36},
                {0X00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

              };
             //Calculo todas las subclaves en la CPU
             for(i=0;i<=3;i++)
                  {
                      for(j=0;j<=3;j++)
                      {
                         matriz_claves[i][j]=clave[i][j];
                      }
                  }
                  for (n=0;n<=9;n++)
                  {
                  for(i=0;i<=3;i++)
                  {
                      for(j=0;j<=3;j++)
                      {
                          sub_claves[i][j]=clave[i][j];
                      }
                  }
                  for (i=0;i<=3;i++)
                  {
                      sub_claves[i][4]=clave[i][3];
                  }
                  temporal=sub_claves[0][4];
                  sub_claves[0][4]=sub_claves[1][4];
                  sub_claves[1][4]=sub_claves[2][4];
                  sub_claves[2][4]=sub_claves[3][4];
                  sub_claves[3][4]=temporal;

                  for(i=0;i<=3;i++)
                  {
                      sub_claves [i][4]=sbox[sub_claves[i][4]];
                  }

                  sub_claves[0][4]=sub_claves[0][0]^sub_claves[0][4]^Rcon[0][n];
                  sub_claves[1][4]=sub_claves[1][0]^sub_claves[1][4]^(0x00);
                  sub_claves[2][4]=sub_claves[2][0]^sub_claves[2][4]^(0x00);
                  sub_claves[3][4]=sub_claves[3][0]^sub_claves[3][4]^(0x00);

                  for(i=0;i<=3;i++)
                  {
                      sub_claves[i][5]=sub_claves[i][1]^sub_claves[i][4];
                      sub_claves[i][6]=sub_claves[i][2]^sub_claves[i][5];
                      sub_claves[i][7]=sub_claves[i][3]^sub_claves[i][6];
                  }
                  for(i=0;i<=3;i++)
                  {
                      clave[i][0]=sub_claves[i][4];
                      clave[i][1]=sub_claves[i][5];
                      clave[i][2]=sub_claves[i][6];
                      clave[i][3]=sub_claves[i][7];
                  }
                  for(i=0;i<=3;i++)
                  {
```

```
                        matriz_claves[i][4*(n+1)]=clave[i][0];
                        matriz_claves[i][4*(n+1)+1]=clave[i][1];
                        matriz_claves[i][4*(n+1)+2]=clave[i][2];
                        matriz_claves[i][4*(n+1)+3]=clave[i][3];
                    }
                    }
//Meto todas las claves en un vector para facilitar las operaciones durante la descodificación

                     for(j=0;j<=10;j++)
                        {
                         for(i=0;i<=3;i++)
                          {
                        vector_claves [16*j+i]=matriz_claves[0][4*j+i];
                        vector_claves[16*j+4+i]=matriz_claves[1][4*j+i];
                        vector_claves[16*j+8+i]=matriz_claves[2][4*j+i];
                        vector_claves[16*j+12+i]=matriz_claves[3][4*j+i];
                          }
                        }
                     //Inicio el contador de tiempo
                     begin = clock();
                     //Comienzo a descodificar los vectores
                                  m=10;
                                  addRoundKey();
                                  for(l=0;l<=8;l++)
                                  {
                                         inv_shiftRows();
                                         inv_subBytes();
                                         m=m-1;
                                         addRoundKey();
                                         inv_mixColumns();
                                  }
                                  inv_shiftRows();
                                  inv_subBytes();
                                  m=m-1;
                                  addRoundKey();

            //Muestro el tiempo de ejecución
            end = clock();
            time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
            printf("El tiempo de ejecucion es %lf \n",time_spent);
//Muestro los vectores descodificados

printf("Los vectores de Test del NIST descodificados son:\n");
              j=0;
              for (i=0;i<64;i++)
                   {
                       printf("%x   ",vector_estado[i]);
                                   j=j+1;
                              if (j==16)
                              {
                                         printf("\n");
                                         j=0;
                              }

                    }

       system ("pause");
       return 0;

}


void addRoundKey ()
```

```
{int i;
//Calculo el XOR entre el estado y la clave
	for (i=0; i<16; i++)
	{
		vector_estado[i]=vector_estado[i]^vector_claves[(16*m)+i];
		vector_estado[i+16]=vector_estado[i+16]^vector_claves[(16*m)+i];
		vector_estado[i+32]=vector_estado[i+32]^vector_claves[(16*m)+i];
		vector_estado[i+48]=vector_estado[i+48]^vector_claves[(16*m)+i];
	}
}
void inv_shiftRows()
{int i;
  unsigned int estado_auxiliar[N];
//Realizo las rotaciones circulares en el sentido opuesto
  for(i=0;i<4;i++)
{
	//Primera fila
	estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
	estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
	estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
	estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
	//Segunda fila
	estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+7];
	estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+4];
	estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+5];
	estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+6];
	//Tercera fila
	estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
	estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
	estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
	estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
	//Cuarta fila
	estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+13];
	estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+14];
	estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+15];
	estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+12];
}
	for(i=0; i<N; i++)

	{
		vector_estado[i]=estado_auxiliar[i];
	}
}
void inv_subBytes()
{	int i;
//Sustituyo cada elemento de la matriz por su equivalente en la matriz de sustitución inversa
//Tabla inversa de sustitucion bytes (iS-box)
    int isbox[256] =     {
    //0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, //0
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, //1
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, //2
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, //3
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, //4
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, //5
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, //6
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, //7
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73, //8
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, //9
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, //A
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, //B
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f, //C
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, //D
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, //E
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};//F
	for(i=0;i<N;i++)
```

```
	{
		vector_estado[i]=isbox[vector_estado[i]];
	}
}
void inv_mixColumns()
{
	int i,j,auxiliar;
	#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))
	//Multiplico cada columna por su inversa en el Campo de Galois
	 // Multiply es un macro que multiplica numeros en el gampo GF(2^8)
	 int num_a,num_b,num_c,num_d;
	 #define Multiply(x,y) (((y & 1) * x) ^ ((y>>1 & 1) * xtime(x)) ^ ((y>>2 &
1) * xtime(xtime(x))) ^ ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^ ((y>>4 & 1) *
xtime(xtime(xtime(xtime(x)))))
		for(i=0;i<4;i++)
		{
			for(j=0;j<4;j++)
			{
num_a = vector_estado[(16*i)+j]; num_b = vector_estado[(16*i+4)+j];
num_c = vector_estado[(16*i+8)+j]; num_d = vector_estado[(16*i+12)+j];
vector_estado[(16*i)+j] = Multiply(num_a, 0x0e) ^ Multiply(num_b, 0x0b) ^
Multiply(num_c, 0x0d) ^ Multiply(num_d, 0x09);
vector_estado[(16*i+4)+j] = Multiply(num_a, 0x09) ^ Multiply(num_b, 0x0e) ^
Multiply(num_c, 0x0b) ^ Multiply(num_d, 0x0d);
vector_estado[(16*i+8)+j] = Multiply(num_a, 0x0d) ^ Multiply(num_b, 0x09) ^
Multiply(num_c, 0x0e) ^ Multiply(num_d, 0x0b);
vector_estado[(16*i+12)+j] = Multiply(num_a, 0x0b) ^ Multiply(num_b, 0x0d) ^
Multiply(num_c, 0x09) ^ Multiply(num_d, 0x0e);
			}
		}
		for(i=0;i<N;i++)
		{
auxiliar=vector_estado[i]/0x100;vector_estado[i]=vector_estado[i]-auxiliar*0x100;
		}
}
```

ANEXO C: Codificado en AES con CUDA

```
#include <stdio.h>
#include <time.h>
#define N 64
//Programa que codifica los 4 vectores de TEST del NIST
//Función que se ejecuta en el device, en este caso calcula AddRoundKey
__device__ void addRounkey (unsigned int*vector_claves, unsigned int*vector_estado,int n)
{
  int i;
  int tID=threadIdx.x;
  for (i=0;i<4;i++)
    {
      if (tID < 16)
    {
      // Cálculo del AddRoundKey

      vector_estado[tID+(16*i)]=vector_estado[tID+(16*i)]^vector_claves[(16*n)+tID];
     }
      __syncthreads();
     }
     }


//Función que calcula la sustitución de bytes, se ejecuta en el device
__device__ void subBytes(unsigned int* vector_estado)
{
```

```
	int tID=threadIdx.x;
// Tabla de sustitución de bytes
int sbox[256] =   {
    //0     1    2      3     4    5     6     7      8    9     A      B    C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};//F
if (tID < N)
 {
	vector_estado[tID]=sbox[vector_estado[tID]];
 }
}
// En shiftRows se rotan las filas, también se ejecutan en la GPU
__device__ void shiftRows (unsigned int* vector_estado)
{
  int tID=threadIdx.x;
  int i;
  unsigned int estado_auxiliar[N];
	for(i=0;i<4;i++)
{
	//Primera fila
	estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
	estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
	estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
	estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
	//Segunda fila
	estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+5];
	estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+6];
	estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+7];
	estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+4];
	//Tercera fila
	estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
	estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
	estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
	estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
	//Cuarta fila
	estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+15];
	estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+12];
	estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+13];
	estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+14];
}
	if (tID < N)
	{
		vector_estado[tID]=estado_auxiliar[tID];
	}
}
// Se multiplica el estado por una matriz en el Campo de Galois, se ejecuta en el
// device
__device__ void mixColumns(unsigned int* vector_estado)
{
	int i;
	int tID=threadIdx.x;
	unsigned char Tmep,Tme,time;
```

```
// xtime es un macro que devuelve el producto con modulo {1b} de {02} y el byte argumento
   #define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))
   for (i=0;i<4;i++)
   {
       if (tID < 4)
       {
        time=vector_estado[(16*i)+tID];
        Tmep = vector_estado[(16*i)+tID] ^ vector_estado[(16*i)+4+tID] ^
vector_estado[(16*i)+8+tID] ^ vector_estado[(16*i)+12+tID] ;
         Tme = vector_estado[(16*i)+tID] ^ vector_estado[(16*i)+4+tID] ;
         Tme = xtime(Tme);
         vector_estado[(16*i)+0+tID] ^= Tme ^ Tmep ;
         Tme = vector_estado[(16*i)+4+tID] ^ vector_estado[(16*i)+8+tID] ;
         Tme = xtime(Tme);
         vector_estado[(16*i)+4+tID] ^= Tme ^ Tmep ;
         Tme = vector_estado[(16*i)+8+tID] ^ vector_estado[(16*i)+12+tID] ;
         Tme = xtime(Tme);
         vector_estado[(16*i)+8+tID] ^= Tme ^ Tmep ;
         Tme = vector_estado[(16*i)+12+tID] ^ time ; Tme = xtime(Tme);
         vector_estado[(16*i)+12+tID] ^= Tme ^ Tmep ;
       }
    }
}
// Es el kernel, la función que se paraleliza
__global__ void aes_funciones_codifica_kernel (unsigned int*vector_estado,
unsigned int*vector_claves)
{
       int l=0;
       int n;
       {
       n=0;
       addRounkey ( vector_claves,  vector_estado, n);
        for(l=0;l<=8;l++)
       {
              subBytes (vector_estado);
              shiftRows (vector_estado);
              mixColumns (vector_estado);
              n=n+1;
              addRounkey ( vector_claves,  vector_estado, n);
        }
         subBytes (vector_estado);
         shiftRows (vector_estado);
         n=n+1;
         addRounkey ( vector_claves,  vector_estado, n);
       }

}
// La función principal del programa, se ejecuta de modo secuencial en la CPU
int main()
{
// Defino las variables que utilizo para calcular el tiempo
    clock_t begin, end;
    double time_spent;
// Variables que se utilizan como contadores
       int i ,j ,n , temporal;
       unsigned int *dev_vector_estado,*dev_vector_claves;
// Reservo los vectores de estado y claves como memoria global
       cudaMalloc ((void**) &dev_vector_estado,N*sizeof(unsigned int));
       cudaMalloc ((void**) &dev_vector_claves,176*sizeof(unsigned int));
```

```
// Tabla de sustitución de bytes
int sbox[256] =   {
    //0     1    2      3     4    5     6     7      8    9     A      B    C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};//F
unsigned int vector_estado[N]={
0x6b,0x2e,0xe9,0x73,0xc1,0x40,0x3d,0x93,0xbe,0x9f,0x7e,0x17,0xe2,0x96,0x11,0x2a,0
xae,0x1e,0x9e,0x45,0x2d,0x03,0xb7,0xaf,0x8a,0xac,0x6f,0x8e,0x57,0x9c,0xac,0x51,0x
30,0xa3,0xe5,0x1a,0xc8,0x5c,0xfb,0x0a,0x1c,0xe4,0xc1,0x52,0x46,0x11,0x19,0xef,0xf
6,0xdf,0xad,0xe6,0x9f,0x4f,0x2b,0x6c,0x24,0x9b,0x41,0x37,0x45,0x17,0x7b,0x10};
//Clave para codificar los vectores de TEST
unsigned int clave[4][4]={
 {0x2b,0x28,0xab,0x09},
 {0x7e,0xae,0xf7,0xcf},
 {0x15,0xd2,0x15,0x4f},
 {0x16,0xa6,0x88,0x3c}
        };
unsigned int matriz_claves[4][44];
//En el vector de claves se almacenan todas las claves,para descodificar las ultimas
16 componentes son la clave de descifrado,es decir, se colocan en sentido inverso
//Una vez calculadas todas las claves las meto en un vector para que sea más
cómodo trabajar en la GPU
unsigned int vector_claves [176];
unsigned int sub_claves[4][8];
int Rcon [4][11]={
                {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36},
                {0X00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

               };
//Calculo todas las subclaves en la CPU
             for(i=0;i<=3;i++)
                  {
                      for(j=0;j<=3;j++)
                      {
                         matriz_claves[i][j]=clave[i][j];
                      }
                  }
                  for (n=0;n<=9;n++)
                  {
                  for(i=0;i<=3;i++)
                  {
                      for(j=0;j<=3;j++)
                      {
                          sub_claves[i][j]=clave[i][j];
                      }
                  }
                  for (i=0;i<=3;i++)
                  {
                      sub_claves[i][4]=clave[i][3];
                  }
```

```
                    temporal=sub_claves[0][4];
                    sub_claves[0][4]=sub_claves[1][4];
                    sub_claves[1][4]=sub_claves[2][4];
                    sub_claves[2][4]=sub_claves[3][4];
                    sub_claves[3][4]=temporal;

                    for(i=0;i<=3;i++)
                    {
                        sub_claves [i][4]=sbox[sub_claves[i][4]];
                    }

                    sub_claves[0][4]=sub_claves[0][0]^sub_claves[0][4]^Rcon[0][n];
                    sub_claves[1][4]=sub_claves[1][0]^sub_claves[1][4]^(0x00);
                    sub_claves[2][4]=sub_claves[2][0]^sub_claves[2][4]^(0x00);
                    sub_claves[3][4]=sub_claves[3][0]^sub_claves[3][4]^(0x00);

                    for(i=0;i<=3;i++)
                    {
                        sub_claves[i][5]=sub_claves[i][1]^sub_claves[i][4];
                        sub_claves[i][6]=sub_claves[i][2]^sub_claves[i][5];
                        sub_claves[i][7]=sub_claves[i][3]^sub_claves[i][6];
                    }
                    for(i=0;i<=3;i++)
                    {
                        clave[i][0]=sub_claves[i][4];
                        clave[i][1]=sub_claves[i][5];
                        clave[i][2]=sub_claves[i][6];
                        clave[i][3]=sub_claves[i][7];
                    }
                    for(i=0;i<=3;i++)
                    {
                        matriz_claves[i][4*(n+1)]=clave[i][0];
                        matriz_claves[i][4*(n+1)+1]=clave[i][1];
                        matriz_claves[i][4*(n+1)+2]=clave[i][2];
                        matriz_claves[i][4*(n+1)+3]=clave[i][3];
                    }
                    }
//Introduzco las claves en el vector_claves para facilitar los cálculos
                     for(j=0;j<=10;j++)
                        {
                         for(i=0;i<=3;i++)
                          {
                        vector_claves [16*j+i]=matriz_claves[0][4*j+i];
                        vector_claves[16*j+4+i]=matriz_claves[1][4*j+i];
                        vector_claves[16*j+8+i]=matriz_claves[2][4*j+i];
                        vector_claves[16*j+12+i]=matriz_claves[3][4*j+i];
                          }
                        }
//Inicio el contador de tiempo
begin = clock();
//Traspaso las variables necesarias a la la GPU
cudaMemcpy(dev_vector_estado, vector_estado, N*sizeof(unsigned int),
cudaMemcpyHostToDevice);
cudaMemcpy(dev_vector_claves, vector_claves, 176*sizeof(unsigned int),
cudaMemcpyHostToDevice);
//Invoco al KERNEL
aes_funciones_codifica_kernel<<<1,N>>>(dev_vector_estado, dev_vector_claves);
//Devuelvo lo calculado en la GPU a la CPU

cudaMemcpy(vector_estado, dev_vector_estado, N*sizeof(unsigned int),
cudaMemcpyDeviceToHost);
//Muestro el tiempo de ejecución
```

```
end = clock();
time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
cudaFree(dev_vector_estado);
cudaFree(dev_vector_claves);
printf("El tiempo de ejecucion es %lf \n",time_spent);
j=0;
for (i=0;i<64;i++)
{
  printf("%x   ",vector_estado[i]);
  j=j+1;
  if (j==16)
  {
   printf("\n");
   j=0;
  }
}
return 0;
}
```

ANEXO D: Descodificado en AES con CUDA

```
#include <stdio.h>
#include<time.h>
#define N 64
//Programa que descodifica los 4 vectores de TEST del NIST
//Calculo el addRoundKey del estado en la GPU
__device__ void addRounkey (unsigned int*vector_claves, unsigned int* vector_estado,int n)
{
int i;
int tID=threadIdx.x;
       for (i=0;i<4;i++)
       {
       if (tID < 16)
       {
       vector_estado[tID+(16*i)]=vector_estado[tID+(16*i)]^vector_claves[(16*n)+tID];
       }
       __syncthreads();
       }
       }
//Función que se ejecuta en el device, en ella se realizan rotaciones circulares
__device__ void inv_shiftRows(unsigned int* vector_estado)
{
  int tID=threadIdx.x;
  int i;
unsigned int estado_auxiliar[N];
for(i=0;i<4;i++)
{
       //Primera fila
       estado_auxiliar[(16*i)+ 0]=vector_estado[(16*i)+0];
       estado_auxiliar[(16*i)+1]=vector_estado[(16*i)+1];
       estado_auxiliar[(16*i)+2]=vector_estado[(16*i)+2];
       estado_auxiliar[(16*i)+3]=vector_estado[(16*i)+3];
       //Segunda fila
       estado_auxiliar[(16*i)+4]=vector_estado[(16*i)+7];
       estado_auxiliar[(16*i)+5]=vector_estado[(16*i)+4];
       estado_auxiliar[(16*i)+6]=vector_estado[(16*i)+5];
       estado_auxiliar[(16*i)+7]=vector_estado[(16*i)+6];
       //Tercera fila
       estado_auxiliar[(16*i)+8]=vector_estado[(16*i)+10];
       estado_auxiliar[(16*i)+9]=vector_estado[(16*i)+11];
       estado_auxiliar[(16*i)+10]=vector_estado[(16*i)+8];
       estado_auxiliar[(16*i)+11]=vector_estado[(16*i)+9];
       //Cuarta fila
```

```
		estado_auxiliar[(16*i)+12]=vector_estado[(16*i)+13];
		estado_auxiliar[(16*i)+13]=vector_estado[(16*i)+14];
		estado_auxiliar[(16*i)+14]=vector_estado[(16*i)+15];
		estado_auxiliar[(16*i)+15]=vector_estado[(16*i)+12];
}
		if (tID < N)
		{
			vector_estado[tID]=estado_auxiliar[tID];
		}
}
//Sustituyo cada elemento del estado por su equivalente en en la S-BOX inversa,
//la función se ejecuta en el device
__device__ void inv_subBytes(unsigned int* vector_estado)
{
int tID=threadIdx.x;
//Tabla inversa de sustitucion bytes (iS-box)
int isbox[256] = {
    //0     1    2     3     4    5     6     7      8    9     A      B    C     D     E     F
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, //0
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, //1
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, //2
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, //3
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, //4
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, //5
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, //6
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, //7
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73, //8
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, //9
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, //A
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, //B
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f, //C
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, //D
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, //E
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d};//F
        if (tID < N)
        {
                vector_estado[tID]=isbox[vector_estado[tID]];
        }
}
//Función que se ejecuta en la GPU, se multiplica cada columna por una matriz en
//el Campo de Galois
__device__ void inv_mixColumns (unsigned int* vector_estado)
{
  int tID=threadIdx.x;
  int i;
  #define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))
//Ronda inversa de Mixcolumn
// Multiply es un macro que multiplica numeros en el gampo GF(2^8)
int auxiliar[N], num_a,num_b,num_c,num_d;
#define Multiply(x,y) (((y & 1) * x) ^ ((y>>1 & 1) * xtime(x)) ^ ((y>>2 & 1) *
xtime(xtime(x))) ^ ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^ ((y>>4 & 1) *
xtime(xtime(xtime(xtime(x))))))

for(i=0;i<6;i++)
{
if(tID<4)
{
num_a = vector_estado[(16*i)+tID];
num_b = vector_estado[(16*i+4)+tID];
num_c = vector_estado[(16*i+8)+tID];
num_d = vector_estado[(16*i+12)+tID];
vector_estado[(16*i)+tID] = Multiply(num_a, 0x0e) ^ Multiply(num_b, 0x0b) ^
Multiply(num_c, 0x0d) ^ Multiply(num_d, 0x09);
vector_estado[(16*i+4)+tID] = Multiply(num_a, 0x09) ^ Multiply(num_b, 0x0e) ^
Multiply(num_c, 0x0b) ^ Multiply(num_d, 0x0d);
```

```
vector_estado[(16*i+8)+tID] = Multiply(num_a, 0x0d) ^ Multiply(num_b, 0x09) ^
Multiply(num_c, 0x0e) ^ Multiply(num_d, 0x0b);
vector_estado[(16*i+12)+tID] = Multiply(num_a, 0x0b) ^ Multiply(num_b, 0x0d) ^
Multiply(num_c, 0x09) ^ Multiply(num_d, 0x0e);
}
}
__syncthreads();
if (tID < N)
{
  auxiliar[tID]=vector_estado[tID]/0x100;
  vector_estado[tID]=vector_estado[tID]-auxiliar[tID]*0x100;
}
}
//Kernel para descodificar los 4 vectores de TEST del NIST, se ejecuta de forma
//paralela en la GPU (device)
__global__ void aes_funciones_descodifica_kernel (unsigned int*vector_estado, unsigned
int*vector_claves)
{
 int n=10;
 int l;
              {
              addRounkey ( vector_claves,  vector_estado, n);
              for(l=0;l<=8;l++)
              {
                     inv_shiftRows(vector_estado);
                     inv_subBytes(vector_estado);
                     n=n-1;
                     addRounkey ( vector_claves,  vector_estado, n);
                     inv_mixColumns (vector_estado);
              }
                     inv_shiftRows(vector_estado);
                     inv_subBytes(vector_estado);
                     n=n-1;
                     addRounkey ( vector_claves,  vector_estado, n);
                     n=10;
              }
}
//Función principal del programa en código C, se ejecuta en la CPU
int main()
{
// Variables que se utilizan para contabilizar el tiempo
clock_t begin, end;
double time_spent;
// Reservo la memoria global para los vectores de claves y estado
unsigned int *dev_vector_estado,*dev_vector_claves;
cudaMalloc ((void**) &dev_vector_estado,N*sizeof(unsigned int));
cudaMalloc ((void**) &dev_vector_claves,176*sizeof(unsigned int));
//Variables que se usan como contadores
int i,j,n,temporal;
// Vector de sustitucion de bytes
int sbox[256] =   {
    //0     1    2      3     4    5     6     7      8    9     A      B    C     D     E     F
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
```

```
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16};//F
//En el vector estado se encuentran los cuatro estados de TEST del NIST para
//descodificar del AES 128bits modo ECB
unsigned int vector_estado[N]=
{0x3a,0x0d,0xa8,0x24,0xd7,0x7a,0x9e,0x66,0x7b,0x36,0xca,0xef,0xb4,0x60,0xf3,0x97,  //Vector 1

0xf5,0x03,0xe7,0x96,0xd3,0xb9,0x85,0xfd,0xd5,0x69,0x89,0xba,0x85,0x9d,0x5a,0xaf,  //Vector 2

0x43,0x59,0x88,0xed,0xb1,0x8e,0x1b,0x03,0xcd,0xce,0x00,0x06,0x7f,0x23,0xe3,0x88,  //Vector 3

0x7b,0x27,0x82,0x04,0x0c,0xe8,0x23,0x72,0x78,0xad,0x20,0x5d,0x5e,0x3f,0x71,0xd4}; //Vector 4
//Clave para descodificar los vectores de TEST
       unsigned int clave[4][4]={
            {0x2b,0x28,0xab,0x09},
            {0x7e,0xae,0xf7,0xcf},
            {0x15,0xd2,0x15,0x4f},
            {0x16,0xa6,0x88,0x3c}
        };
unsigned int matriz_claves[4][44];
//En el vector de claves se almacenan todas las claves, para descodificar las
ultimas 16 componentes son la clave de descifrado, es decir, se colocan en
sentido inverso.Una vez calculadas todas las claves las meto en un vector para
que sea más cómodo trabajar en la GPU
        unsigned int vector_claves [176];
        unsigned int sub_claves[4][8];
        int Rcon [4][11]={
                {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36},
                {0X00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
                {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
              };
             //Calculo todas las subclaves en la CPU
             for(i=0;i<=3;i++)
                  {
                      for(j=0;j<=3;j++)
                      {
                         matriz_claves[i][j]=clave[i][j];
                      }
                  }
                  for (n=0;n<=9;n++)
                  {
                  for(i=0;i<=3;i++)
                  {
                      for(j=0;j<=3;j++)
                      {
                          sub_claves[i][j]=clave[i][j];
                      }
                  }
                  for (i=0;i<=3;i++)
                  {
                      sub_claves[i][4]=clave[i][3];
                  }
                  temporal=sub_claves[0][4];
                  sub_claves[0][4]=sub_claves[1][4];
                  sub_claves[1][4]=sub_claves[2][4];
                  sub_claves[2][4]=sub_claves[3][4];
                  sub_claves[3][4]=temporal;
                  for(i=0;i<=3;i++)
                  {
                      sub_claves [i][4]=sbox[sub_claves[i][4]];
                  }
                  sub_claves[0][4]=sub_claves[0][0]^sub_claves[0][4]^Rcon[0][n];
                  sub_claves[1][4]=sub_claves[1][0]^sub_claves[1][4]^(0x00);
```

```
                    sub_claves[2][4]=sub_claves[2][0]^sub_claves[2][4]^(0x00);
                    sub_claves[3][4]=sub_claves[3][0]^sub_claves[3][4]^(0x00);
                    for(i=0;i<=3;i++)
                    {
                        sub_claves[i][5]=sub_claves[i][1]^sub_claves[i][4];
                        sub_claves[i][6]=sub_claves[i][2]^sub_claves[i][5];
                        sub_claves[i][7]=sub_claves[i][3]^sub_claves[i][6];
                    }
                    for(i=0;i<=3;i++)
                    {
                        clave[i][0]=sub_claves[i][4];
                        clave[i][1]=sub_claves[i][5];
                        clave[i][2]=sub_claves[i][6];
                        clave[i][3]=sub_claves[i][7];
                    }
                    for(i=0;i<=3;i++)
                    {
                        matriz_claves[i][4*(n+1)]=clave[i][0];
                        matriz_claves[i][4*(n+1)+1]=clave[i][1];
                        matriz_claves[i][4*(n+1)+2]=clave[i][2];
                        matriz_claves[i][4*(n+1)+3]=clave[i][3];
                    }
                    }
                   //Introduzco las claves en el vector
                     for(j=0;j<=10;j++)
                        {
                         for(i=0;i<=3;i++)
                          {
                        vector_claves [16*j+i]=matriz_claves[0][4*j+i];
                        vector_claves[16*j+4+i]=matriz_claves[1][4*j+i];
                        vector_claves[16*j+8+i]=matriz_claves[2][4*j+i];
                        vector_claves[16*j+12+i]=matriz_claves[3][4*j+i];
                          }
                        }
//Inicio el contador de tiempo
begin = clock();
//Traspaso las variables necesarias a la la GPU
cudaMemcpy(dev_vector_estado, vector_estado, N*sizeof(unsigned int),
cudaMemcpyHostToDevice);
cudaMemcpy(dev_vector_claves, vector_claves, 176*sizeof(unsigned int),
cudaMemcpyHostToDevice);
//Invoco al KERNEL
{
aes_funciones_descodifica_kernel<<<1,N>>>(dev_vector_estado, dev_vector_claves);
}
//Devuelvo lo calcualdo en la GPU a la CPU
cudaMemcpy(vector_estado, dev_vector_estado, N*sizeof(unsigned int),
cudaMemcpyDeviceToHost);
//Muestro el tiempo de ejecución
end = clock();
time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
printf("El tiempo de ejecucion es %lf \n",time_spent);
printf("Los vectores de Test del NIST descodificados son:\n");
j=0;
for (i=0;i<64;i++)
{
  printf("%x   ",vector_estado[i]);
  j=j+1;
   if (j==16)
       {
         printf("\n");
         j=0;
```

```
        }
}
system ("pause");
cudaFree(dev_vector_estado);
cudaFree(dev_vector_claves);
return 0;
}
```

Referencias Bibliográficas

[1] **RAE.** Diccionario de la lengua española (22º edición, 2001), Real Academia Española.

[2] **INTECO**. Observatorio de la seguridad de la información. *La criptografía desde la antigüedad a la máquina enigma*. Cuaderno de notas del observatorio. [En línea] 22 de junio de 2014. https://www.inteco.es/file/q65GjNMApk_6gqdnJquPKw

[3] **ITESCAM**. *Historia de la criptografía*. [En línea] 22 de junio de 2014. www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r68645.**PDF**

[4**] Jorge Ramió Aguirre** (1 de marzo de 2006). *Libro Electrónico de Seguridad Informática y criptografía*. Departamento de Publicaciones de la Escuela Universitaria de Informática de la Universidad Politécnica de Madrid. [En línea] 23 de junio de 2014. www.criptored.upm.es/guiateoria/gt_m001a.htm

[5] **Facultad de Ingeniería**. Universidad Nacional Autónoma de México. *Historia de la Criptografía - Discos, cilindros y otros métodos de cifrado*. [En línea] 23 de junio de 2014. http://redyseguridad.fi-p.unam.mx/proyectos/criptografia/criptografia/index.php/2-tecnicas-clasicas-de-cifrado/22-opereraciones-utilizadas/222-algoritmos-de-sustitucion?showall=&start=5

[6] **Kriptópolis** (21 de julio de 2013). *Un reto con el criptógrafo de Wheatstone con clave homofónica.* [En línea] 23 de junio de 2014. http://www.kriptopolis.com/criptografia-wheatstone

[7] **Pablo G. Bejerano** (2014, 6 de febrero). *Código Enigma, descifrado: el papel de Turing en la Segunda Guerra Mundial*. El Diario.es. [En línea] 24 de junio de 2014. www.eldiario.es/turing/.../alan-turing-**enigma**-codigo_0_226078042.html

[8] **Departamento de Matemática Aplicada**. *Criptosistema de clave pública. El cifrado RSA.* Facultad de Informática. Universidad Politéctica de Madrid. [En línea] 24 de junio de 2014. www.dma.fi.upm.es/java/matematicadiscreta/aritmeticamodular/**rsa**.html

[9] **Herramientas Web para la enseñanza de protocolos de comunicación**. *DES.* [En línea] 24 de junio de 2014. http://neo.lcc.uma.es/evirtual/cdd/tutorial/presentacion/**des**.html

[10] **Eduardo Bonilla Palencia** (2012). *Implementación del algoritmo AES sobre arquitectura ARM con mejoras en rendimiento y seguridad*. (Proyecto Fin Carrera). Universidad Carlos III de Madrid. [En línea] 24 de junio de 2014. http://e-archivo.uc3m.es/handle/10016/15402

[11] **CCN** (24 de agosto de 2009). *Guía de seguridad de las TIC, Glosario y Abreviaturas.* [En línea] 24 de junio de 2014. https://www.ccn-cert.cni.es/publico/serieCCN.../es/c/cryptosystem.htm

[12] **José Luis Prieto** (4 de noviembre de 2013). *Criptosistema.* [En línea] 25 de junio de 2014.

http://jlprietofsi1314.blogspot.com.es/2013/11/criptosistema.html

[13] **David Rodríguez Sánchez** (2009). *Control y auditoría de coreos electrónicos en Lotus Notes*, Capítulo 3.5 págs. 55-63. (Proyecto Fin de Carrera). Universidad Carlos III de Madrid. [En línea] 25 de junio de 2014. http://e-archivo.uc3m.es/handle/10016/8525

[14] **Manuel José Lucena López** (mayo 2003). *Criptografía y Seguridad en Computadores,* Conceptos Básicos sobre Criptografía págs. 29-34 y 126-130 [En línea] 28 de junio de 2014 www.uned.es/413042/material/**Criptografia**.pdf

[15] **Mohit Arora** (5 de julio de 2012). *How secure is AES against brute force attacks?* [En línea] 29 de junio de 2014. http://www.eetimes.com/document.asp?doc_id=1279619

[16] **Joan Daemen** y **Vicent Rijmen** (1998). *The Design of Rijndael, AES-The Advanced Encryption Standard.* (ISBN 3-540-42580-2).

[17] **Francisco Pais Suárez** (4 de febrero de 2003). *Estudio del algoritmo de cifrado Rijndael. Comparativa entre los algoritmos de cifrado DES y Rijndael.* (Proyecto Fin de Carrera). Universidade da Coruña [En línea] 1 de julio de 2014. www.academia.edu/7105052/Proyecto-rinjdael

[18] **Aldo Jiménez Arteaga** (2014). *Criptografía – Criptografía de Clave Secreta*. [En línea] 2 de julio de 2014. http://samhain.softgot.com/criptografia/lecturasnotas.html

[19] **NIST** (26 de noviembre de 2001). Federal Information Processing Standards Publication 197. *Announcing the Advanced Encryption Standard (AES).* [En línea] 1 de agosto de 2014. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[20] **Enrique Zabala** (2008). *The Rijndael Animation y Rijndael Iinspector.* [En línea] 1 de agosto de 2014. http://www.formaestudio.com/rijndaelinspector/

[21] **InCON Team** (5 de septiembre de 2008). *List of test vectors for AES ECB 128-bit encryption mode.* [En línea] 1 de agosto de 2014. http://www.inconteam.com/software-development/41-encryption/55-aes-test-vectors#aes-cbc-128

[22] **INTECO**. *Legislación nacional*. [En línea] 5 de agosto de 2014. www.inteco.es/Formacion/Legislación/

[23] **BOE** (14 de diciembre de 1999). *Ley Orgánica 15/1999, del 13 de diciembre, de Protección de Datos de Carácter Personal*. [En línea] 5 de agosto de 2014. www.boe.es/buscar/act.php?id=BOE-A-1999-23750

[24] **BOE** (10 mayo de 2014). *Ley 9/2014, de 9 de mayo, de Telecomunicaciones*. [En línea] 5 de agosto de 2014. www.boe.es/diario_boe/txt.php?id=BOE-A-2014-4950

[25] **BOE** (12 de julio de 2002). *Ley 34/2002, de 11 de julio, de servicios de la sociedad de la información y de comercio electrónico.* [En línea] 5 de agosto de 2014. www.boe.es/buscar/act.php?id=BOE-A-2002-13758

[26] **BOE** (31 de diciembre de 2011). *Real Decreto Legislativo 1/1996, de 12 de abril, por el cual se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y*

amortizando las disposiciones legales vigentes sobre la materia. [En línea] 5 de agosto de 2014. www.boe.es/buscar/act.php?id=BOE-A-1996-8930

[27] **Pablo Jara Werchau y Patricia Nazar**. *Estándar IEEE 802.11x de las WLAN pág. 3*. Editorial de la Universidad Tecnológica Nacional. [En línea] 6 de agosto 2014. www.edutecne.utn.edu.ar/monografias/standard_802_11.pdf

[28**] Carlos Hernández Chacón** (12 de noviembre del 2009). *Criptoanálisis práctico de WEP y WAP sobre WLAN 802.11*, Capítulo 1.2.8.2 pág. 7. (Trabajo Fin de Carrera). Universidad Politécnica de Catalunya. [En línea] 6 de agosto de 2014. *https://upcommons.upc.edu/pfc/bitstream/2099.1/7865/1/memoria.pdf*

[29] **IETF** (diciembre 2005). *Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)*. [En línea] 6 de agosto de 2014. http://tools.ietf.org/html/rfc4309

[30] **Intel.** *Tecnología Intel Data Protection con AES-NI y Secure Key*. [En línea] 6 de agosto de 2014. http://www.intel.es/content/www/es/es/architecture-and-technology/advanced-encryption-standard--aes-/data-protection-aes-general-technology.html

[31] **Abelardo Palomino Guzmán, Ángel Manuel Romero Zamora y Alfonso Solbes Bosch** (curso 2005/2006). *Diseño e implementación de algoritmos criptográficos sobre FPGA*, Capítulo 1.3 págs. 9-12 Proyectos de Sistemas Informáticos. Universidad Complutense de Madrid. [En línea] 6 de agosto de 2014. http://eprints.ucm.es/8911/1/Memoria_de_Proyecto.pdf

[32] **Rubén Lumbiarres-López, Mariano López-García y Enrique F, Cantó-Navarro.** *Ataques por canal lateral sobre el algoritmo de encriptación AES implementado en MicroBlaze*. [En línea] 6 de agosto de 2014. http://www.researchgate.net/publication/261333052_Ataques_por_canal_lateral_sobre_el_algoritmo_de_encriptacin_AES_implementado_en_MicroBlaze

[33] **A.R.** (17 de agosto de 2011). *El algoritmo de encriptación AES, más vulnerable de lo que se creía. El País. [En línea] 6 de agosto de 2014.* http://sociedad.elpais.com/sociedad/2011/08/17/actualidad/1313532009_850215.html

[34] **Microsoft Developer Network**. *Introducción a Visual Studio.* [En línea] 7 de agosto de 2014. http://msdn.microsoft.com/es-es/library/fx6bk1f4(v=vs.90).aspx

[35] **Laboratorio del Departamento de Informática.** *Microsoft Academic Alliance*. [En línea] 7 de agosto de 2014. http://www.lab.inf.uc3m.es/servicios/msdnaa

[36] **José Antonio Gómez Ruiz.** *Introducción al Lenguaje C/C++*. Asignatura Fundamentos de Informática. ETSI Industrial. Universidad de Málaga. [En línea] 8 de agosto de 2014. www.lcc.**uma**.es/~janto/ftp/fundinf/trans_t3.pdf

[37] **Carlos Alberto Fernández y Fernández** (19 de octubre de 1998). *Lenguaje C de programación. Una breve introducción*. Instituto de Electrónica y computación. Universidad Tecnológica de la Mixteca. [En línea] 8 de agosto de 2014. www.utm.mx/~caff/progEstruc/**lenguajeC**.pdf

[38] **Qt Project** (22 de enero de 2014). *Qt Creator*. [En línea] 8 de agosto de 2014. http://qt-project.org/wiki/Category:Tools::QtCreator_Spanish

[39] **Eduardo Bonilla Palencia** (mayo 2012). *Código PFC Implementación del algoritmo AES sobre arquitectura ARM con mejoras en rendimiento y seguridad*. (Proyecto Fin de Carrera) Universidad Carlos III de Madrid. [En línea] 10 de agosto de 2014. http://e-archivo.uc3m.es/handle/10016/15402?locale-attribute=en

[40] **Stackoverflow** (9 de marzo de 2011). *Execution time of C program.* [En línea] 12 de agosto de 2014. http://stackoverflow.com/questions/5248915/execution-time-of-c-program

[41] **Alfredo G. Escobar Portillo** (enero 2011). *Sistema de programación matemática en paralelo empleando tarjetas gráficas*, Capítulo 3 págs. 41-68 (Proyecto Fin de Carrera). Universidad Pontificia Comillas. [En línea] 15 de agosto de 2014. www.iit.upcomillas.es/pfc/resumenes/4dee5b199a840.pdf

[42] **Nvidia**. *¿Qué es el GPU computing?* [En línea] 16 de agosto de 2014. http://www.nvidia.es/object/gpu-computing-es.html

[43] **Nvidia**. *GeForce GTX 550 Ti*. [En línea] 17 de agosto de 2014. http://www.nvidia.es/object/product-geforce-gtx-550ti-es.html

[44] **Nvidia**. *Procesamiento paralelo CUDA*. [En línea] 17 de agosto de 2014. http://www.nvidia.es/object/cuda-parallel-computing-es.html

[45] **Manuel Ujaldón**. *CUDA 6.0*. Universidad de Málaga. [En línea] 17 de agosto de 2014. http://gpu.cs.uct.ac.za/Slides/**CUDA6**.pdf

[46] **Lucas Manuel Rodríguez y Pablo Odorico** (septiembre 2012). *Una introducción a GPGPU con OpenCL.* Escuela Argentina de GPGPU Computing para aplicaciones científicas. [En línea] 17 de agosto de 2014. http://dl.ea**gpgpu**.org/c1/**clases**/c1_**clase1**.pdf

[47] **Alfonso García Pérez, Guillermo Hernández González y Daniel Tabas Madrid** (curso 2008-2009). *Paralelización con CUDA de algoritmos de verificación facial*, Capítulo 3 págs. 15-35. (Proyecto Fin de Carrera). Universidad Complutense de Madrid. [En línea] 15 de agosto de 2014. http://eprints.ucm.es/9456/

[48] **Manuel Ujaldón** (febrero 2014). *Nvidia CUDA Fellow.* [En línea] 17 de agosto de 2014. http://dc.exa.unrc.edu.ar/**rio2014**/sites/default/files/1_**CUDA**-1.pdf

[49**] Keisure Iwai y Takakazu Kurokawa** (2010). *AES encryption implementation on CUDA GPU and its analysis*. National Defense Academy of Japan.

[50] **Qinjian Li y Chengwen Zhong** (2012). *Implementation and Analysis of AES Encryption on GPU.* Northwestern Polytechnical University Xi'an, China.

[51] **Bhupathi Kakarlapudi y Nitin Alabur.** *Comparison of Hardware Implementations of S-box and T-box architectures of AES*. [En línea] 20 de agosto de 2014.

http://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCEQFjAA&url=http%3A%2F%2Fteal.gmu.edu%2Fcourses%2FECE746%2Fproject%2Fslides_2008%2FAES_T-box_slides.pdf&ei=5BoWVI2iCdDpaJbDgrAJ&usg=AFQjCNF2lITd7mXAHNGylIz-pLJCbLFUhA

[52] **Enrique Arias Antúnez y José Luis Sánchez García.** *Computación de altas prestaciones con GPUs*. Presente y futuro de los sistemas de computación. Cursos de verano 2010. [En línea] 21 de agosto de 2014. *www.dsi.uclm.es/siscomp/material/presentacion_GPUs.ppt*

[53] **David Capello** (1 de marzo de 2008). *Medir el tiempo de una rutina*. [En línea] 22 de agosto de 2014. http://dacap.com.ar/blog/cpp/medir-el-tiempo-de-una-rutina/

[54] **Nvidia**. *Nvidia Jetson TK1. Embedded Development Kit.* [En línea] 2 de septiembre de 2014. http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html